МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Московский физико-технический институт
(государственный университет)

В.Я. Митницкий

АРХИТЕКТУРА IBM PC И ЯЗЫК АССЕМБЛЕРА

*Рекомендовано Учебно-методическим советом
Московского физико-технического института
(государственного университета)
в качестве учебного пособия
для студентов высших учебных заведений
по направлению "Прикладные математика и физика"*

МОСКВА 2000

УДК 681.325.5
ББК 32.973
М66

Рецензенты:
Кафедра математической физики Московского педагогического государственного университета
Доктор физико-математических наук, профессор *С.Д. Якубович*

Митницкий В.Я.
М66 Архитектура IBM PC и язык Ассемблера: Учеб. пособие. — М.: МФТИ, 2000. — 148 с.
ISBN 5-7417-0136-1

Рассмотрены общая структура программы на Ассемблере, команды двоичной арифметики, работа компьютера в текстовом и графическом режимах, реализация на Ассемблере различных конструкций языков высокого уровня, принципы отображения структур данных на двоичный код. На конкретных примерах разъясняются общие идеи архитектуры современных ЭВМ.
Для студентов и преподавателей вузов, а также для всех желающих познакомиться с нижним уровнем программирования современных микропроцессорных систем.

УДК 681.325.5
ББК 32.973

ISBN 5-7417-0136-1

Предисловие

Зачем нужен Ассемблер?

Степень мотивации при изучении того или иного раздела науки или практического навыка имеет первостепенное значение и во многом определяет успех или неудачу. Для себя вопрос можно сформулировать более точно: зачем *мне* нужно изучать Ассемблер?

При честном ответе самому себе на этот вопрос, несомненно, необходимо учитывать соображения научной и практической значимости проблемы, честолюбия (в хорошем смысле), практических последствий (получение диплома, возможность устройства на работу), стремление к постижению научной истины и, далеко не в последнюю очередь, материальные, финансовые интересы.

В настоящее время на всем земном шаре миллионы, если не миллиарды, людей вовлечены в процесс компьютеризации. Без преувеличения можно сказать, что люди, общающиеся с компьютером на уровне Ассемблера, стоят на вершине этой гигантской пирамиды. Это — жрецы информатики, они же относятся к самым высокооплачиваемым программистам. Таким образом, изучив Ассемблер, можно в полной мере удовлетворить собственное честолюбие не в ущерб окружающим и приобрести специальность с высокой степенью востребованности.

Переходя к более специфическим аспектам, необходимо отметить, что программа на Ассемблере как никакая другая близка к машинному коду, является лишь "слегка причесанным" машинным кодом.

Отсюда в качестве постулата можно констатировать, что ни один язык программирования высокого уровня, и даже ни один какой-либо программный продукт вообще, не может обеспечить создание более эффективной и гибкой программы, чем Ассемблер. Все, что можно создать при помощи любых программных продуктов, можно создать и при помощи Ассемблера. Обратное неверно. Правда, за это приходится платить снижением производительности труда при программировании.

Именно Ассемблер (и ничто другое) служит инструментом для построения интерфейсов и поддержки протоколов связи с любыми нетрадиционными внешними устройствами (например, с аппаратурой некоторого физического эксперимента, космического корабля и т. п.)

Неоспорима методическая ценность изучения Ассемблера для досконального понимания работы компьютера. Полностью и до конца понять работу компьютера (за исключением его электроники) можно, только освоив Ассемблер и управление компьютером на самом нижнем уровне.

Можно утверждать, что знание Ассемблера и умение работать с компьютером на самом нижнем уровне обязательны для системного программиста-профессионала. Однако и ученый-прикладник может делать простые ассемблерные вставки в свои программы на Паскале или на С++, "вылизывая" наиболее ответственные их места. Кстати, можно и наоборот, вставлять процедуры на языках высокого уровня в ассемблерную программу.

Наконец, нельзя не отметить удовольствие от программирования, когда вы работаете один на один с "железом" компьютера и между вами и компьютером никто не стоит (имеется в виду автор компилятора с языка высокого уровня), как бы ни был высок его профессиональный уровень.

Итак, если вы хотите до конца понять, как работает ваш компьютер и использовать возможности компьютера на 100 %, создавать интерфейсы с внешними устройствами, перехватчики, перекодировщики, надежно защищать свои программные продукты и иметь потенциальную возможность взламывать чужие и многое, много другое, — этот курс для вас.

* * *

Автор выражает благодарность члену-корреспонденту РАН В. П. Иванникову, К. Б. Бухарову и М. В. Ивановскому, прочитавшим рукопись и сделавшим ряд ценных замечаний.

1. Архитектурные особенности процессоров Intel™

1.1. Представление целых чисел в процессоре и в памяти

1.1.1. Применяемые системы счисления

Натуральное n-значное число в любой позиционной системе счисления с основанием Z записывается следующим образом:

$$\alpha_n \alpha_{n-1} \ldots \alpha_1 = \sum_{i=1}^{n} \alpha_i Z^{i-1}$$, где α_i — одна из цифр.

В частности, повсеместно принятая десятичная (decimal) система счисления своим широким распространением обязана лишь тому факту, что у человека на руках 10 пальцев. В этой системе натуральное n-значное число записывается в виде:

$$\alpha_n \alpha_{n-1} \ldots \alpha_1 = \sum_{i=1}^{n} \alpha_i (10)^{i-1}$$, где α_i одна из арабских цифр 0..9.

Очевидны следующие факты:

- Для Z-ичной системы счисления необходимы ровно Z цифр.
- Максимальное натуральное n-значное число в Z-ичной системе счисления равно Z^{n-1}, т. е. единице с n нулями минус единица (например, максимальное двузначное десятичное число 99=100–1).
- Сложение, вычитание, умножение и деление натуральных чисел в любой системе счисления производятся аналогично ("в столбик", как учили в школе).
- Умножение числа на Z^k сводится к сдвигу числа на k разрядов влево.
- Деление числа (нацело) на Z^k сводится к сдвигу числа на k разрядов вправо.
- Перевод числа из одной системы счисления в другую состоит в приравнивании соответствующих сумм и рассмотрении цифр числа в новой системе счисления как неопределенных коэффициентов в сумме (такой перевод осуществляется единственным образом).

Память компьютера состоит из электронных элементов (ячеек), каждый из которых может находиться (при включенном компьютере) в одном из двух устойчивых состояний, одному из которых условно приписывается значение 0, а другому — 1. По-

этому для хранения чисел в памяти компьютера естественно использовать **двоичную** (**b**inary) систему счисления:

$$\alpha_n \alpha_{n-1} \ldots \alpha_1 = \sum_{i=1}^{n} \alpha_i 2^{i-1}, \text{ где } \alpha_i \text{ — 0 или 1.}$$

Будем отмечать двоичные числа буквой ***b*** после числа, а десятичные числа — буквой ***d*** или не отмечать никак (такие обозначения приняты и в Ассемблере). Например: 101*b*=5*d*=5.

Очевидны следующие факты:

- Максимальное натуральное *n*-значное число в двоичной системе счисления равно $2^n - 1$ (например, максимальное трехзначное двоичное число 111*b*=2^3-1).
- Общее число всех различных целых неотрицательных *n*-значных чисел в двоичной системе равно 2^n (включая 0).
- Умножение числа на 2^k сводится к сдвигу числа на *k* разрядов влево.
- Деление числа (нацело) на 2^k сводится к сдвигу числа на *k* разрядов вправо.

При программировании на Ассемблере необходимо уметь переводить числа из десятичной в двоичную систему и наоборот.

Перевод из двоичной системы в десятичную систему производится прямым подсчетом приведенной выше суммы.

Перевод из десятичной системы в двоичную производится при помощи простого алгоритма: исходное число делится на ближайшую степень двойки, не превосходящую этого числа, остаток от деления делится на следующую (меньшую) степень двойки и т. д. Частные, получаемые от таких делений, будут последовательно (слева направо) давать разряды числа в двоичном представлении.

Однако двоичное представление неудобно при записи чисел — числа получаются очень длинными. Поэтому при записи чисел в программе чаще используется **шестнадцатеричная** (**h**exadecimal) система счисления. В ней вводятся 6 новых "цифр" при помощи латинских букв: A=10, B=11, C=12, D=13, E=14, F=15. (малые или большие латинские буквы — безразлично). Будем отмечать шестнадцатеричные числа буквой ***h*** после числа. Кроме того, шестнадцатеричное число всегда должно начинаться с

арабской цифры (чтобы отличить его от идентификатора). Иначе перед числом ставят незначащий ноль, например: 0E7*h*.

Шестнадцатеричные числа очень легко переводятся в двоичные и наоборот. Для этого достаточно заметить, что каждая шестнадцатеричная цифра равна некоторому четырехзначному двоичному числу (тетраде).

Кроме того, в Ассемблере довольно редко применяется **восьмеричная** (octal) запись чисел. Будем отмечать восьмеричные числа буквами ***o*** или ***q*** после числа (***o*** слишком похоже на ноль), например, 73*q*.

Восьмеричные числа очень легко переводятся в двоичные и наоборот. Для этого достаточно заметить, что каждая восьмеричная цифра равна некоторому трехзначному двоичному числу (триаде).

Ниже приводятся некоторые неотрицательные целые числа в различных системах счисления.

decimal	hex	octal	binary
0	0*h*	0*q*	0*b*
1	1*h*	1*q*	1*b*
2	2*h*	2*q*	10*b*
3	3*h*	3*q*	11*b*
4	4*h*	4*q*	100*b*
5	5*h*	5*q*	101*b*
6	6*h*	6*q*	110*b*
7	7*h*	7*q*	111*b*
8	8*h*	10*q*	1000*b*
9	9*h*	11*q*	1001*b*
10	0A*h*	12*q*	1010*b*
11	0B*h*	13*q*	1011*b*
12	0C*h*	14*q*	1100*b*
13	0D*h*	15*q*	1101*b*
14	0E*h*	16*q*	1110*b*
15	0F*h*	17*q*	1111*b*
16	10*h*	20*q*	10000*b*
$2^8-1=255$	0FF*h*	377*q*	11111111*b*
$2^{16}-1=65535$	0FFFF*h*	177777*q*	1111111111111111*b*

Можно представлять себе, что в каждый момент работы компьютера его оперативная память (RAM) "забита" двоичными

числами (как говорят, двоичным кодом). **Этот двоичный код** изменяется со временем.

1.1.2. Биты, байты, слова, двойные слова

Одна ячейка памяти, которая может находиться в состоянии 0 или 1, называется битовой ячейкой, или просто **битом (bit)**.

Можно считать, что в памяти компьютера битовые ячейки вытянуты в одну линию, начало которой находится слева или сверху, а конец — справа или снизу (такова традиция изображения на рисунках и схемах). С начала этой линии соседние 8 битовых ячеек объединяются и составляют один **байт (byte)**. Байты нумеруются слева направо (или сверху вниз), начиная с нулевого байта. Номер байта называется его **адресом** в памяти. Внутри байта биты нумеруются справа налево от 0 (правый младший бит) до 7 (старший левый бит). Нумерация битов внутри байта условна и никакого отношения к адресу не имеет.

Поскольку программирование осуществляется при помощи адресов, программист может работать только с байтами, а не с битами, так как у последних адресов нет. Этот важный факт выражается следующей фразой: **байт есть наименьшая адресуемая часть памяти**. Как следует из предыдущего раздела, в байте можно закодировать 256 различных целых неотрицательных чисел (от 0 до 255).

Два любых соседних байта могут быть объединены в **слово (word)**. Адресом слова считается адрес его левого (верхнего) байта. Как следует из предыдущего раздела, в слове можно закодировать 2^{16} =65536 различных целых неотрицательных чисел (от 0 до 65535). Внутри слова биты нумеруются справа налево от 0 (правый младший бит) до 15 (старший левый бит). Поскольку слово состоит из двух байт, то байт, содержащий разряды от 0 до 7, называется младшим, байт, содержащий разряды от 8 до 15, называется старшим.

Четыре любых соседних байта могут быть объединены в **двойное слово (double word)**. Адресом двойного слова считается адрес его левого (верхнего) байта. Как следует из предыдущего раздела, в двойном слове можно закодировать 2^{32} различных целых неотрицательных чисел. Внутри двойного слова биты нумеруются справа налево от 0 (правый младший бит) до 31

(старший левый бит). Поскольку двойное слово состоит из двух слов, то слово, содержащее разряды от 0 до 15, называется младшим, слово, содержащее разряды от 16 до 31, называется старшим.

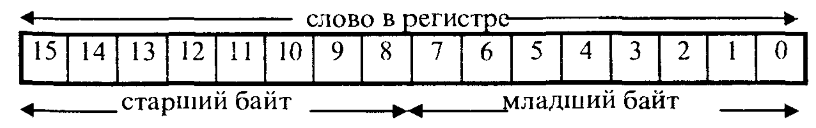
Все процессоры Intel аппаратно поддерживают работу с байтами, словами и двойными словами.

Отсюда следует важный факт: программист, работая с адресом, должен указать не только этот адрес, но и **тип** переменной (т. е. тем или иным способом указать, имеет ли он в виду байт, слово или двойное слово).

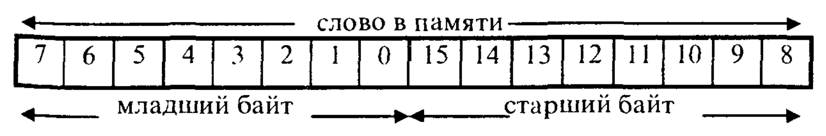
Кроме того, в процессоре Intel имеется несколько именованных 16-битовых (2-байтовых) ячеек памяти, называемых **регистрами**. (В процессорах 386 и старше некоторые регистры расширены до 4-байтовых, что мы сейчас во внимание принимать не будем). В некоторых регистрах также могут храниться байты и слова.

Необходимо отметить одну крайне неприятную особенность хранения слов в памяти, сложившуюся исторически с тех времен, когда регистры процессоров были 8-разрядными.

Слово в регистре процессора хранится в "нормальном" виде, т. е. старший байт слева, младший байт справа:



Однако слово в памяти хранится в "перевернутом" виде, т. е. младший байт слева, старший байт справа:



Итак, все процессоры Intel требуют размещения данных по принципу: **младший байт по младшему адресу**.

Эта "перевернутость" слов в памяти автоматически учитывается всеми командами Ассемблера, если обращаться к слову как слову, а к двойному слову как к двойному слову. Однако если,

например, необходимо считать только старший байт слова, эту особенность необходимо учесть и увеличить адрес на 1. Эту особенность необходимо иметь в виду также, например, при анализе распечатки (дампа) некоторой области памяти.

1.1.3. Беззнаковые и знаковые величины. Прямой и дополнительный коды

Итак, мы рассмотрели, как хранятся целые неотрицательные числа (**беззнаковые величины**) в диапазоне 0..255 (в байтах) и в диапазоне 0..65535 (в словах). А как хранить числа, которые могут быть как положительными, так и отрицательными (**знаковые величины**)?

Для знака числа выделяется самый старший бит (7-й) в байте или самый старший бит в слове (15-й), причем считается, что если знаковый бит равен 0, то число положительное (точнее неотрицательное), а если знаковый бит равен 1, то число отрицательное.

Казалось бы, можно сделать просто: если знаковый бит равен 1, то к двоичному числу в младших семи битах (для байта) приписывается знак "минус" и аналогично для слова. Однако сделано не так.

Отрицательное число записывается в так называемом **дополнительном коде** к соответствующему положительному числу. Преобразование к дополнительному коду состоит в инвертировании значений всех битов и в прибавлении к полученному числу 1. Например:

5*d*=00000101*b*; –5*d*=11111010*b*+1*b*=11111011*b*

Легко проверить, что все законы арифметики при этом остаются в силе. Проверим, например, что

(+5)+(–5)=0. Действительно,

```
  00000101
+ 11111011
  00000000
```

Что дает применение дополнительного кода для отрицательных чисел? Оказывается, что операции сложения и вычитания остаются одними и теми же независимо от того, считаем ли мы операнды беззнаковыми или знаковыми. Действительно, предыду-

щий числовой пример можно трактовать и как сложение двух беззнаковых двоичных чисел (правда, с вытеснением получающегося 9-го разряда за разрядную сетку). На любых других числовых примерах можно убедиться, что, во-первых, выполняются законы арифметики отрицательных чисел, и, во-вторых, выполняются законы арифметики беззнаковых чисел. Проверим еще, например, числовое равенство для знаковых величин 7–4=3 или (+7)+(–4)=3:

7=00000111*b*; 4=00000100*b*; –4=11111011*b*+1*b*=11111100*b*

```
  00000111
+ 11111100
  00000011 =3d
```

Эту же операцию можно трактовать и как сложение двух беззнаковых 8-разрядных величин.

Конечно, дополнительный код отрицательного числа можно было бы получить вычитанием из нуля соответствующего положительного числа, однако инвертирование и прибавление 1 осуществить технически проще.

Все то же самое, что сказано о байтах, можно повторить для слов.

Выясняется на первый взгляд парадоксальная ситуация: и беззнаковые и знаковые величины записываются *одним и тем же* двоичным кодом, к которому применимы одни и те же операции сложения и вычитания, и только *отношение программиста* к этому коду должно быть различным. Он должен учитывать, что представление знаковых величин сдвигается в сторону отрицательных значений: от –128 до +127 для байтов и от –32768 до +32767 для слов. Кроме того, к сожалению, операции умножения и деления для беззнаковых и знаковых величин различаются.

1.1.4. Двоично-десятичная кодировка (BDC)

Специально для взаимодействия с внешней аппаратурой, работающей обычно с десятичными данными, в процессорах Intel предусмотрена поддержка двоичной кодировки десятичных цифр в двух форматах.

Неупакованный формат: Десятичная цифра закодирована двоичным кодом в младшем полубайте. Значения бит в старшем полубайте во внимание не принимаются.

Упакованный формат: В каждом полубайте двоичным кодом закодирована одна десятичная цифра. Цифра старшего полубайта является старшей.

1.2. Реальный режим (80x86)

1.2.1. Понятие о реальном и защищенном режимах

Процессоры Intel до 286 могли работать только в однозадачном режиме, который назвыается реальным режимом или режимом 80x86. Начиная с процессора 286. введена возможность многозадачного режима работы, основной проблемой которого была защита данных каждой задачи от всех остальных. Такой режим будем называть защищенным. Следует отметить, что процессор 286 с этой задачей не справился и являлся по существу более быстрой версией 80x86. Только на процессорах 386 и выше с введением новых регистров и повышением разрядности старых регистров удалось достигнуть уверенной работы в защищенном режиме.

Важно отметить, что реальный режим (80x86) работы сохраняется для всех высших моделей процессоров.

В настоящем пособии мы в основном рассмотрим практическое программирование для реального режима и лишь наметим основные идеи, заложенные в осуществлении защищенного режима.

1.2.2. Регистры процессора и их назначение

Любой процессор для работы в реальном режиме имеет четырнадцать 16-разрядных ячеек сверхбыстрой памяти, называемых регистрами. Они делятся на две группы и два отдельных регистра: **группа регистров общего назначения** (РОН) — 8 регистров, **группа сегментных регистров** — 4 регистра, регистр флагов и регистр указателя инструкций. Название группы "регистры общего назначения" не должно вводить в заблуждение — регистры в ней не равноправны, каждый регистр имеет свое специфическое назначение и может использоваться в посторонних целях только тогда, когда он свободен от выполнения своих

"прямых обязанностей". Иногда групну регистров общего назначения подразделяют на **регистры данных** (AX, BX, CX, DX) и **регистры указателей** (SI, DI, BP, SP). Иногда регистры BX, SI, DI и BP называют **регистрами-модификаторами**.

Имя	English	Основное назначение	Прим.
		РЕГИСТРЫ ОБЩЕГО НАЗНАЧЕНИЯ	
AX	Accumulator	Основной сумматор	AH,AL
BX	Base	Адресация по базе	BH,BL
CX	Counter	Счетчик циклов	CH,CL
DX	Data	Для "длинных" данных	DH,DL
SI	Source Index	Индексирование источника	
DI	Destination Index	Индексирование приемника	
BP	Base Pointer	База стека	
SP	Stack Pointer	Вершина стека	
		СЕГМЕНТНЫЕ РЕГИСТРЫ	
CS	Code Segment	Сегмент команд, не может быть изменен напрямую	
SS	Stack Segment	Сегмент стека	
DS	Data Segment	Сегмент данных	
ES	Extension Segment	Дополнительный сегмент	
		РЕГИСТР ФЛАГОВ	
FLAGS	Flags	Информация о текущем состоянии процессора	
		РЕГИСТР УКАЗАТЕЛЯ КОМАНД	
IP	Instruction Pointer	Смещение команды, программно недоступен	

Более точное назначение регистров будет проясняться по мере изучения команд, здесь приведем лишь некоторые общие данные. К первым четырем регистрам общего назначения можно обращаться как целиком (например, AX), так и к их старшим (AH) или младшим (AL) байтам. Регистр SP используется исключительно для указания на вершину стека. Значения регистров CS и IP не могут быть изменены непосредственно в программе (они изменяются либо автоматически, либо косвенно за счет команд переходов). Имя регистра IP вообще не является зарезервированным и не может встречаться в программе (в смысле имени регистра).

В регистре флагов имеют определенный смысл лишь 0-й, 2-й, 4-й биты и биты с 6 по 11. По значениям некоторых флагов (флагов состояния) можно судить о результатах выполнения команд, о возникновении ошибок. Другие флаги (управляющие), наоборот, задают некоторые режимы работы процессора.

Биты регистра флагов перечислены в следующей таблице:

Бит	Имя	Название	Тип	Назначение
0	CF	Carry Flag	сост.	Перенос или заем
2	PF	Parity Flag	сост.	Четность
4	AF	AuxiliaryFlag	сост.	Перенос или заем BDC
6	ZF	Zero Flag	сост.	Ноль результата
7	SF	Sign Flag	сост.	Знак результата
8	TF	Trace Flag	упр.	Трассировка
9	IF	Interrupt Flag	упр.	Разрешение прерываний
10	DF	DirectionFlag	упр.	Напр. обработки цепочек
11	OF	Overflow Flag	сост.	Переполнение

1.2.3. Развитие архитектуры процессора Intel

Описанная в предыдущем разделе архитектура процессора характерна для моделей до 80286 включительно. В компьютер мог устанавливаться так называемый арифметический сопроцессор 80x87 для аппаратной поддержки арифметики чисел с плавающей точкой. Работа с двойными словами аппаратно не поддерживалась.

Начиная с модели 80386, процессор Intel был существенно модернизирован. Основная цель этой — модернизации поддержка надежной работы защищенного режима, однако возможности реального режима были также существенно расширены (в частности, введена аппаратная поддержка двойных слов). Все регистры общего назначения расширены до 32 разрядов и получили наименования EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP (буква E в названии означает Extended — расширенный). Программист получил возможность работать как с полным 32-разрядным регистром (например, ESI), так и с его младшим словом (SI). Кроме того, сохранилась возможность работать со

старшим и младшим байтами младшего слова первых четырех регистров (например, DH и DL).

Сегментные регистры сохранены 16-разрядными, но введены два новых дополнительных регистра FS и GS. Их роль полностью аналогична роли старого сегментного регистра ES.

Регистры флагов и указателя команд также были расширены до 32 разрядов и получили имена EFLAGS и EIP.

Начиная с процессора 80486, арифметический сопроцессор был интегрирован с основным процессором, в результате чего последний получил 11 новых регистров (FPU/NPX) для поддержки арифметики с плавающей точкой.

Начиная с процессора Pentium *MMX*, в стандарт процессора вводится расширение MMX (MultiMedia eXtension) для упрощения обработки больших потоков данных, связанных в основном со звуком и изображением. Идея MMX состоит в использовании регистров арифметики с плавающей точкой для аппаратной поддержки учетверенных (64-битных) слов. Для MMX введена также новая арифметика "с насыщением". Например, если результат байтовой операции превышает 255, в режиме насыщения он полагается равным 255. Далее вводятся расширения уже к базовому набору команд MMX, например, AMD 3D.

1.2.4. Адресное пространство процессора. Физические адреса, сегменты и смещения

Адресным пространством процессора назовем множество адресов, которое в состоянии генерировать данный процессор. Адресное пространство может не совпадать с реальной памятью, поскольку, с одной стороны, ячейки, соответствующие некоторым адресам, могут отсутствовать, и, с другой стороны, некоторые ячейки памяти могут быть недоступны для адресации (что часто бывает).

Допустим, что в некотором регистре мы сформировали адрес байта памяти. Поскольку регистр 16-разрядный, таким образом можно адресовать не более 2^{16} =65536 байт = 64 Кбайт (1 Кбайт=1024 байт), что явно недостаточно.

Последовательно занумеруем все байты в памяти, начиная с нуля, и назовем номер каждого байта **физическим адресом байта**. Очевидно, физические адреса слишком велики, чтобы поместиться в регистр процессора.

Мысленно разобьем всю память на **параграфы**. Параграфами назовем последовательные участки памяти по 16 байт. 1-й параграф будет простираться от 0*h* до 0F*h*, 2-й — от 10*h* до 1F*h*, 1-й — от 20*h* до 2F*h* и т. д. Важно заметить, что физический адрес первого байта в начале каждого параграфа оканчивается на шестнадцатеричную цифру 0*h* (или, что то же самое, на 0000*b*).

Адресное пространство процессора 80x86 расширяется за счет сегментной адресации. **Сегментом** называется сплошная область памяти размером не более 64 Кбайт, начало которой совпадает с началом какого-либо параграфа. Иначе говоря, физический адрес первого байта сегмента должен быть кратен 16. Поэтому последняя шестнадцатеричная цифра физического адреса сегмента не будет нести никакой полезной информации — она заведомо будет нулем. Назовем **сегментной частью адреса (или просто сегментом — segment)** физический адрес начала сегмента с откинутым шестнадцатеричным нулем. (Здесь есть некоторая путаница: сегментом называют и область памяти и численную характеристику ее начала, подобно тому, как иногда называют сопротивлением и сам элемент, и величину сопротивления, которое он оказывает электрическому току).

Положение отдельного байта внутри сегмента будем называть **смещением (offset)**. Смещение есть номер отдельного байта, отсчитанный от начала сегмента. Комбинацию сегмента и смещения называют просто **адресом** байта в памяти и записывают двумя 4-значными шестнадцатеричными цифрами через двоеточие: СЕГМЕНТ:СМЕЩЕНИЕ, например, 0B000:1F4E.

Восстановить физический адрес по сегменту и смещению очень просто: необходимо приписать справа ноль к значению сегмента (или, что то же самое, умножить на 16, или, что то же самое, сдвинуть на одну шестнадцатеричную позицию влево) и затем прибавить значение смещения. Пример: **1234*h*:5678*h***

Физический адрес: **12340**

+ 5678

179B8 *h* = 96696 *d*

Необходимо отметить, что два сегмента могут совпадать, частично перекрываться, располагаться впритык друг к другу и, наконец, располагаться полностью отдельно.

Соответственно, один и тот же байт памяти может адресоваться многими способами, например, 0100*h*:0050*h*, 0102*h*:0030*h* и 0103*h*:0020*h* дают один и тот же физический адрес 1050*h*.

Не надо думать, что все сегменты имеют размер 64 Кбайт — чаще всего они намного меньше (но не меньше 16 байт).

Посчитаем объем адресного пространства процессора, имея в виду, что значения сегмента и смещения записаны в двух 16-разрядных регистрах процессора. Очевидно, максимально возможный адрес есть 0FFFF*h*:0FFFF*h*. Соответствующий физический адрес 0FFFF0*h*+0FFFF*h*= 10FFEF*h*=1113095 байт = 1М+64К–17. Таким образом, объем адресного пространства процессора 8086 приблизительно равен 1 Мбайт и 64 Кбайт.

Здесь уместно привести пример того, как адресное пространство может не совпадать с реальной физической памятью. Как известно, архитектура IBM PC включает в себя концепцию шины. Шина включает в себя линии данных, адресные линии и линии управления. Когда процессор хочет связаться с некоторым внешним для него устройством (и с памятью в том числе) и, например, передать данные, он выставляет эти данные на линиях данных шины, а адрес устройства выставляет на адресных линиях шины. Если контроллер памяти устанавливает, что выставленный адрес принадлежит адресному пространству памяти, он записывает данные, выставленные на линиях данных шины в соответствующую ячейку памяти.

В ранних моделях IBM PC было всего 20 адресных линий шины. Поэтому адресные линии шины могли пропустить не более 2^{20} =1 Мбайт адресов. Поэтому на таких машинах объем физической адресуемой памяти был не более 1 Мбайт, а верхние 64 Кбайт памяти использовать было невозможно, хотя процессор и позволял это. На более поздних моделях при работе в реальном режиме можно использовать все 1 М+64 К–17 байт памяти.

Таким образом, для указания адреса некоторой ячейки памяти (или адреса устройства) необходимо указать оба элемента адреса: сегмент и смещение. Это делается следующим образом. Смещение указывается в команде (машинной или ассемблер-

ной) либо непосредственно, либо указывается имя регистра или ячейки памяти, содержащих это смещение. Способ, которым указывается смещение в машинной или ассемблерной команде, называется **способом адресации**. Сегмент адреса находится в одном из сегментных регистров. В каком именно? Имеются правила умолчания для определения сегментного регистра при формировании данного адреса. Однако если эти правила не устраивают программиста, имя сегментного регистра может быть указано явно.

1.2.5. Типичная схема адресного пространства 80x86

В приведенной ниже таблице показана типичная схема адресного пространства 80x86 при работе под управлением MS DOS (адреса возрастают сверху вниз).

Адрес	Область памяти	Объем	Класс памяти
00000	Векторы прерываний	1 К	Обычная память (640 К) Conventional memory
00400*h*	Данные BIOS	256 б	
00500*h*	MS DOS	ок. 80 К	
	Транзитная область	ок. 550 К	
0A0000	Графический видеобуфер	64 К	Верхняя память (384 К) Upper memory (UM)
0B0000	Транзитная область	32 К	
0B8000	Текстовый видеобуфер	32 К	
0C0000	Транзитная область	192 К	
0F0000	BIOS	64 К	
100000	Верхняя память High Memory Area	64 К	HMA
10FFF0	Расширенная память Extended /Expanded Memory Specification	до 4 Г	XMS/EMS

Первые 640 К памяти с адресами с 00000*h* по 9FFFF*h* называются обычной памятью (conventional memory). В ней последовательно располагаются векторы прерываний (256 векторов по 4 байта), содержащие ссылки на подпрограммы обработки прерываний, область данных BIOS (базовой системы ввода/вывода), включающую в себя, в частности, буфер клавиатуры и адреса портов. Далее располагается сама операционная система MS DOS, занимающая около 80 Кбайт. Далее от конца MS DOS до адреса 0A0000*h* располагается транзитная область

памяти, в которой пользователь размещает необходимые ему резидентные программы и запускает свои программы. Размер транзитной области равен 640 Кбайт минус все перечисленное выше, т. е. около 500–600 Кбайт. Это вся память, которой может распоряжаться пользователь по своему усмотрению, не принимая каких-либо специальных мер по расширению памяти даже в том случае, когда в компьютере есть еще свободные физические блоки памяти.

Отсюда следует, что каждый килобайт обыкновенной памяти крайне ценен, и существуют специальные методы экономии этой памяти. В частности, в верхней памяти (UM), простирающейся от адреса 0A0000*h* до 100000*h* также имеются транзитные области — так называемые блоки верхней памяти (UMB). Для работы с этими свободными адресами необходимо, во-первых, иметь физические блоки свободной памяти, и, во-вторых, загрузить менеджер верхней памяти EMM386 или его аналоги *(QEMM)*. Для работы с блоком *HMA* необходимо загрузить менеджер верхней памяти *HIMEM*. В этих блоках можно располагать необходимые пользователю резидентные программы (например, русификатор, драйвер мыши, кэш-буфер дисков SMARTDRV), для их загрузки в UMB в операционной системе для файлов CONFIG.SYS и AUTOEXEC.BAT предусмотрены специальные директивы DEVICEHIGH и LOADHIGH. Можно также загрузить в верхнюю память значительную часть MS DOS, доведя размер транзитной области обыкновенной памяти до 620 Кбайт.

2. Язык программирования Ассемблера

2.1. Процесс программирования и выполнения программы

Программированием мы называем составление программы, т. е. списка инструкций (команд), которые должен будет последовательно выполнять процессор. Этот список инструкций в виде двоичного кода должен быть размещен в памяти компьютера. Выполнение программы начинается тогда, когда процессор будет настроен на первую выполняемую команду программы. Настройка производится следующим образом: сегмент первой выполняемой команды загружается в регистр сегмента команд CS, а смещение первой команды загружается в регистр указателя команд IP. После этого первая команда выполняется, а в регистр IP автоматически заносится смещение следующей команды, подлежащей выполнению (точнее, значение *IP* увеличивается на длину выполненной команды). Обычно команды выполняются в естественной последовательности, т. е. в порядке их написания в программе, но иногда эта последовательность может нарушаться, а именно тогда, когда среди команд встречается команда перехода (иногда называют также командой передачи управления). Такие команды фактически изменяют содержимое регистра IP (а иногда и CS). Явным образом изменить содержимое этих регистров невозможно. После выполнения всех инструкций программы управление передается на другую программу (например, в операционную систему, которая также является программой), т. е. фактически производится перенастройка IP и CS. Следует отметить, что процессор никогда не простаивает (при работе под управлением *DOS*).

Процессор воспринимает команды только в двоичном коде. Соответственно составление программы в двоичном (машинном) коде является самым нижним уровнем, на котором возможно программирование. Однако программирование в машинном коде является крайне трудоемким. С другой стороны, программирование на любом языке верхнего уровня нуждается в компиляции, т. е. в переводе конструкций этого языка в машинный код.

Язык Ассемблера является языком программирования самого нижнего уровня. В него введены лишь минимальные удобства

для программиста: возможность использования имен переменных, меток для переходов и мнемокодов вместо кодов операций.

Итак, в комплект программных продуктов для разработки ассемблерных программ должен входить компилятор и компоновщик. Поскольку за логикой ассемблерной программы проследить гораздо труднее, чем за логикой программы на языках высокого уровня, третьим неотъемлемым компонентом инструментария для разработки ассемблерных программ является отладчик, дающий возможность пошагового выполнения программы, наблюдения за значениями регистров и ячеек памяти, остановки выполнения в фиксированных местах программы и т. п.

Существует несколько комплектов программных продуктов для разработки ассемблерных программ. Наиболее популярными из них являются компилятор, компоновщик и отладчик TASM, TLINK и TD компании Borland, а также MASM, LINK и CV компании Microsoft. Сами языки Ассемблера TASM и MASM также несколько различаются, однако в большинстве случаев удается создавать программы, переносимые с одного стандарта на другой.

Программа на Ассемблере пишется при помощи любого текстового редактора (необходимо следить лишь за тем, чтобы редактор не вставлял в файл дополнительную информацию по форматированию текста и т. п.) и помещается в файл с любым именем и расширением ASM. Такой файл называется исходным модулем. Затем запускается программа компилятора Ассемблера и в качестве аргумента ей передается имя исходного модуля. Если компиляция прошла без ошибок, создается новый файл с тем же именем и расширением OBJ, называемый объектным модулем. Затем запускается программа компоновщика, предназначенная для разрешения внешних ссылок в объектном модуле и для объединения объектных модулей, если их несколько. В качестве аргумента программы компоновщика ей передается имя объектного модуля (или имена нескольких модулей). В результате работы компоновщика при отсутствии ошибок порождается абсолютный модуль — готовая к выполнению программа в файле с расширением EXE или COM.

Задавая различные опции компилятора и отладчика, можно попутно получать или не получать файл листинга (с расшире-

нием LST), содержащий протокол компиляции, и файл распределения памяти (с расширением MAP).

Можно написать простой командный файл для последовательной компиляции ассемблерной программы, ее компоновки и выполнения полученного исполняемого файла, например, для компилятора TASM:

```
echo off
tasm  -zi  -l  %1
if errorlevel 1 goto exit
tlink  -v  -x  %1
if errorlevel 1 goto exit
%1
:exit
```

Здесь ключ компилятора **zi** служит для включения в obj-файл информации для работы отладчика, ключ **l** разрешает генерацию файла листинга. Ключ компоновщика **v** служит для включения в exe-файл информации для работы отладчика, ключ **x** подавляет генерацию map-файла.

2.2. Синтаксис ассемблерной программы

При описании различных конструкций языка будем придерживаться следующих соглашений. Лексемы, написанные с применением русского языка (за исключением комментариев) и выделенные курсивом, означают обобщенные понятия, нуждающиеся в конкретных значениях. Лексемы, выделенные жирным шрифтом, являются обязательными, лексемы, не выделенные жирным шрифтом, могут быть опущены. Например, формат команды MOV:

Метка: **MOV** **Операнд1, Операнд2** *;Комментарий*

Алфавит Ассемблера включает большие и малые буквы латинского алфавита, символ подчеркивания (который также считается буквой), цифры и спецсимволы. По умолчанию различий между большими и малыми буквами не делается, однако можно задать опцию компилятора, при которой большие и малые буквы будут различаться. Буквы кириллицы допустимы только в комментариях и в значениях символьных переменных и констант.

Лексемами Ассемблера могут быть ключевые слова, имена (идентификаторы), вводимые пользователем, и числа. Имя

должно состоять из латинских букв и цифр, причем первым символом имени должна быть буква. Внутри имени не должно быть пробелов и прочих разделителей.

Программа на Ассемблере состоит из **команд и директив**. Различие между директивами и командами следующее: при компиляции команда непосредственно порождает команду для процессора, директива не порождает команд процессора, а служит для описания переменных, задания режимов работы компилятора и т. п.

Каждая команда или директива пишется с отдельной строчки и состоит из четырех необязательных полей, отделяемых друг от друга любым количеством пробелов.

Формат команды:
Метка: **Мнемокод** *Операнды* *;Комментарий*

Формат директивы:
Имя **Мнемокод** *Операнды* *;Комментарий*

При набивке программы удобно отделять одно поле от другого знаком табуляции, тогда программа будет располагаться в удобочитаемом виде в четыре колонки (хотя это и не обязательно).

2.3. Простейшая программа

Напишем простейшую программу, выводящую на экран сообщение и неформально объясним, как она работает. Формальное определение используемых конструкций будет дано позднее.

```
assume SS:MySg1,DS:MySg2,CS:MySg3
              ;Соответствие сегментов и сегментных регистров
MySg1   segment stack        ;Начало сегмента стека
        db 128 dup(?)        ;Стек размером 128 байт
MySg1   ends                 ;Конец сегмента стека
MySg2   segment              ;Начало сегмента данных
Msg     db  "It works$"      ;Строка сообщения
MySg2   ends                 ;Конец сегмента данных
MySg3   segment              ;Начало сегмента команд
Entry:                       ;Метка точки входа в программу
mov     AX,MySg2             ;Адрес сегмента данных в AX
mov     DS,AX                ;Настройка сегм. регистра данных
```

```
12  mov    AH,09h       ;Номер функции 21-го прерывания
13  lea    DX,Msg       ;В DX адрес строки сообщения
14  int    21h          ;Вывод на экран
15  mov    AH, 4Ch      ;Номер функции 21-го прерывания
16  int    21h          ;Передача управления в DOS
17  MySg3  ends         ;Конец сегмента команд
18  end    Entry        ;Конец программы с точкой входа Entry
```

Стоящие в левой колонке номера строк частью программы не являются и введены только для облегчения дальнейшего объяснения программы.

Для понимания этой программы необходимо отметить, что в любой программе на Ассемблере необходимо выполнить некоторые действия, не связанные непосредственно с задачей, которую выполняет данная программа, своего рода "накладные расходы". Эти действия связаны в основном с организацией сегментной адресации.

Любая программа на Ассемблере в простом случае состоит чаще всего из трех программных сегментов: сегмента команд, сегмента данных и сегмента стека. Поэтому в каждой программе необходимо, вообще говоря, выполнить следующие действия:

1. Определить начала и концы программных сегментов команд, данных и стека (и может быть, других программных сегментов).
2. Указать соответствие программных сегментов и сегментных регистров, при помощи которых будет производиться адресация.
3. Загрузить в сегментные регистры адреса начал соответствующих программных сегментов.

Некоторые из этих действий выполняются автоматически или автоматически по умолчанию, однако некоторые действия необходимо указывать явно. С учетом этих замечаний приступим к построчному анализу программы.

Строка 1: Этой директивой мы указываем сегментные регистры, которые будут применяться по умолчанию при адресации в трех сегментах нашей программы, т. е. по существу указываем,

что регистр MySg1 будет сегментом стека, MySg2 — сегментом данных и MySg3 — сегментом команд.

Строка 2: Директива SEGMENT открывает новый программный сегмент с именем MySeg1, который в соответствии с содержанием директивы ASSUME есть сегмент стека. Сегмент стека должен быть определен в программе, даже если он фактически не используется (как в данном случае), и его размер должен быть не менее 128 байт. Параметр STACK дает возможность не загружать в регистр SS адрес начала сегмента стека, это будет выполнено автоматически.

Строка 3: При помощи директивы DB выделяем под стек 128 байт без их инициализации.

Строка 4: Директива ENDS закрывает программный сегмент с именем MySeg1.

Строка 5: Директива SEGMENT открывает новый программный сегмент с именем MySeg2, который в соответствии с содержанием директивы ASSUME есть сегмент данных.

Строка 6: Директива DB определяет переменную Msg как строку байт и инициализирует ее. Символ $ означает конец строки.

Строка 7: Директива ENDS закрывает программный сегмент с именем MySeg2.

Строка 8: Директива SEGMENT открывает новый программный сегмент с именем MySeg3, который в соответствии с содержанием директивы ASSUME есть сегмент команд.

Строка 9: Метка первой выполняемой команды программы. Мы могли бы исключить эту строку и пометить меткой Entry непосредственно следующую строку.

Строки 10–11: Служат для загрузки адреса начала сегмента данных в сегментный регистр DS. Сначала загружаем имя сегмента данных MySg2 (а фактически его адрес) в регистр AX, а затем из регистра AX в регистр DS. Сразу возникают два вопроса. Почему это нельзя сделать одной командой? Потому что сегментные регистры можно загрузить только из регистров общего назначения и никаким другим образом. Почему мы таким же образом не загружаем регистры CS и SS? Потому что регистр команд загружается автоматически при запуске програм-

мы из DOS, то же самое происходит с регистром стека, когда в директиве определения программного сегмента стека указан параметр STACK.

Строки 12–13: Подготовка к вызову 21-го прерывания для вывода сообщения на экран. Для взаимодействия с внешними устройствами (и с видеокартой в том числе) имеются процедуры DOS, оформленные в виде прерываний. Мы будем пользоваться в основном прерываниями 10*h* и 21*h*. Каждое из этих прерываний представляет собой целое собрание различных процедур, снабженных номерами. При возникновении необходимости взаимодействия с внешним устройством нужно загрузить в регистр AH номер требуемой процедуры, в другие регистры загрузить данные, необходимые для работы требуемой процедуры, и, наконец, вызвать необходимое прерывание. В данном случае мы загружаем в регистр AH номер 09*h* процедуры вывода строки на экран, а в регистр DX — смещение начала строки сообщения (значение сегмента будет взято из сегментного регистра DS). Команда LEA действует аналогично команде MOV, но загружает в DX не значение переменной, а ее адрес (т. е. смещение ее первого байта).

Строка 14: Команда INT вызывает прерывание с указанным номером. В данном случае по этому прерыванию осуществляется вывод строки на экран.

Строка 15: Подготовка к вызову 21-го прерывания для завершения программы. Завершение любой программы есть по существу передача управления в другую программу (например, в DOS). Такая передача управления также выполняется при помощи функции 4C*h* прерывания 21*h*.

Строка 16: Вызов 21-го прерывания для завершения программы и передачи управления в DOS.

Строка 17: Директива ENDS закрывает программный сегмент с именем MySeg3.

Строка 18: Директива END завершения программы. В качестве операнда должна быть указана метка точки входа.

Эта же программа может быть оформлена короче, при помощи упрощенных директив сегментации:

```
        .model   small              ;Задание модели памяти
        .stack   128                ;Определение сегмента стека
        .data                       ;Начало сегмента данных
Msg     db   "It works!$"
        .code                       ;Начало сегмента команд
Entry:  mov  AX,@data     ;В регистр AX адрес сегм. данных
        mov  DS, AX
        mov  AH, 09h
        lea  DX,Msg
        int  21h
        mov  AH, 4Ch
        int  21h
        end  Entry
```

Здесь комментарии даны только к строкам, отличающимся от соответствующих строк предыдущего варианта.

В чем отличие этого варианта программы от предыдущего? Здесь мы не вводим имен программных сегментов и не применяем парных директив SEGMENT-ENDS для обозначения начал и концов сегментов. Здесь применяются упрощенные директивы сегментации программы (.DATA, .CODE и .STACK), каждая из которых действует до появления следующей аналогичной директивы. Отпадает надобность в директиве ASSUME, поскольку по умолчанию за сегментом команд закрепляется регистр CS, за сегментом данных — DS, за сегментом стека — SS. Кроме того, размер стека теперь можно задать, как операнд директивы .STACK. Для загрузки адреса сегмента данных в регистр DS (через AX) применяется специальный стандартный идентификатор @DATA (применяемый вместе с директивой .DATA).

Однако здесь необходимо задать одну из семи моделей памяти, которая будет использоваться нашей программой (директива .MODEL в строке 1). В дальнейшем мы всегда будем использовать модель памяти SMALL, означающую, что все данные будут находиться в единственном сегменте данных, а команды — в единственном сегменте команд.

Таким образом, можно написать "скелет" простой программы на Ассемблере, внутрь которого можно добавлять необходимый код и данные:

```
        .model  small
        .stack  128             ;При необходимости добавить!
        .data
; СЮДА ВСТАВИТЬ ОПИСАНИЕ ВСЕХ ДАННЫХ
        .code
Entry:  mov     AX,@data
        mov     DS,AX
; СЮДА ВСТАВИТЬ НЕОБХОДИМЫЙ КОД
        mov     AH,4Ch
        int     21h
        end     Entry
```

2.4. "Препроцессорные" директивы INCLUDE и EQU

Директивы INCLUDE и EQU аналогичны директивам препроцессорной обработки #include и #define языка С. В отличие от большинства директив и команд Ассемблера эти две директивы могут стоять где угодно в программе, в том числе вне всех программных сегментов.

Формат директивы INCLUDE: **include *ИмяФайла***

Операнд *ИмяФайла* записывается по правилам операционной системы (и может включать в себя имя диска и имена каталогов).

Директива действует следующим образом. В то место, где стоит директива, вписывается указанный файл (если, конечно, он доступен). Фактически на компиляцию поступает текст программы со включенным файлом.

Если мысленно представить, что все директивы INCLUDE в программе заменены на соответствующие им файлы, результирующая программа должна быть *правильной* программой на Ассемблере.

При помощи директивы INCLUDE можно оформить "скелет" простой программы на Ассемблере так, что не будет необходимости вмешиваться в него и вписывать что-либо вручную.

```
        .model       small
        .stack       256                    ; С запасом!
        .data
include      my_data.asm
```

```
; ВКЛЮЧЕНИЕ ФАЙЛА ОПРЕДЕЛЕНИЯ ДАННЫХ
        .code
Entry:  mov   AX,@data
        mov   DS,AX
include  my_codes.asm
; ВКЛЮЧЕНИЕ ФАЙЛА С КОДАМИ ПРОГРАММЫ
        mov   AH,4Ch
        int   21h
        end   Entry
```

Такой шаблон можно поместить в файл и назвать его, например, EXESHELL.ASM, поскольку такая структура исходного модуля приведет в итоге к созданию исполняемого EXE-файла (исходный модуль для создания COM-файла должен иметь другую структуру).

Теперь необходимо только в текущем каталоге (в каталоге, из которого производится запуск компилятора) подготовить файл определения данных конкретной программы MY_DATA.ASM и файл кодов конкретной программы MY_CODES.ASM и запустить компилятор.

Директива EQU также может стоять в любом месте программы, в том числе вне любого программного сегмента. Ее формат:

Имя **equ** ***Операнд***

Все три поля директивы являются обязательными. Смысл директивы состоит в том, что во всем тексте программы ниже директивы любое вхождение имени заменяется на имя или значение операнда.

Если в качестве операнда в правой части написано также некоторое имя, то имя, указанное в левой части, становится его псевдонимом, т. е. появляются два имени-синонима, которые можно использовать ниже по программе в равной степени для обозначения одного и того же объекта.

Если в качестве операнда в правой части написано некоторое константное выражение, то оно вычисляется, и ниже по программе каждое вхождение имени из левой части заменяется на это вычисленное значение.

Если в качестве операнда в правой части написана числовая константа, то директива является просто определением именованной константы (на практике чаще всего используется именно этот слу-

чай, например, для задания числа элементов массива). Значение именованной константы, заданной директивой EQU, не может быть изменено при помощи другой директивы EQU.

Если в качестве операнда в правой части написан некий текст, который нельзя отнести ни к чему перечисленному выше, имя в правой части будет просто сокращенным обозначением текста.

Примеры:

```
counter  equ   CX
N        equ   2*K-1
maxbyte  equ   0FFh
nofile   equ   "File not found"
```

2.5. Директивы описания и инициализации переменных DB, DW и DD

2.5.1. Переменная и ее атрибуты

С точки зрения программирования переменная есть некоторая непрерывная область оперативной памяти, которой присвоено символическое имя. Каждая переменная, как и в языках высокого уровня, имеет три атрибута: **имя, значение и тип**. Рассмотрим их подробнее.

Имя — удобный для запоминания программиста псевдоним адреса первого байта области памяти, выделенной под переменную.

Значение — двоичный код, хранящийся в данной области памяти По мере выполнения программы значение может изменяться.

Тип — длина переменной в байтах, иначе, объем области памяти, выделенной под переменную. Напомним, что по некоторому адресу может быть расположен байт, слово или двойное слово, поэтому если тип измерять в байтах, он может иметь значения 1, 2 и 4. Тип переменной однозначно определяет диапазон целых чисел, которые могут храниться в данной переменной. Например в байте могут быть записаны целые числа в диапазоне от -128 до 255, в слове могут быть записаны целые числа в диапазоне от -32768 до 65535.

В Ассемблере имеется специальный оператор **type**, который, будучи применен к уже определенной переменной, возвращает

значение 1, 2 или 4. Например, если выше в программе определена переменная P типа WORD, то значение выражения TYPE P равно 2.

Чтобы работать с некоторой переменной в Ассемблере, ее необходимо сначала создать (или определить).

При определении некоторой переменной (статической) компилятору во всех случаях необходимо сообщить имя и тип создаваемой переменной. Можно также сообщить компилятору начальное значение этой переменной, как говорят, **инициализировать** ее, но это не обязательно (переменная может принять определенное значение позднее).

Особый подход в Ассемблере применен к определению массивов, т. е. к совокупности некоторого числа однотипных элементов. Память выделяется для всех элементов массива, а имя (т. е. фактически адрес) определяется только для первого элемента массива. Далее программист может обращаться к некоторому элементу массива только "кустарно", самостоятельно вычисляя адрес элемента по его сдвигу в памяти относительно первого элемента, и по адресу первого элемента.

2.5.2. Выражения

Выражения применяются в Ассемблере, но они имеют **совершенно иной смысл**, нежели выражения в языках высокого уровня. Основой для понимания этого различия служит тот факт, что в языках высокого уровня выражения вычисляются **во время выполнения программы**, а в Ассемблере выражения вычисляются **во время компиляции**. Выражения бывают *константные* и *адресные*.

Константное выражение может содержать числа (в диапазоне от -2^{15} до $2^{16}-1$), одиночные символы или пары символов (рассматриваемые как числа), константы, определенные по директиве EQU, и арифметические и некоторые другие операции с ними. Значениями константных выражений являются двухбайтные целые числа. Характерной особенностью константного выражения является то, что программист, в принципе, вместо этого выражения может написать в программе его значение (вычислив, например, это значение на калькуляторе).

Адресное выражение может содержать метки команд, имена переменных и некоторые операции с ними, например, сложение с целым числом или вычитание целого числа. Значениями адресных выражений являются двухбайтные адреса (по существу, смещения). Адресное выражение можно перевести в константное при помощи оператора OFFSET, т. е. выражение **offset *Адр_Выражение*** будет уже константным.

Допустим, что в программе определена переменная *X* и затем где-то встретилось выражение *X*+1. Ни в коем случае нельзя это понимать, как увеличение кода в ячейке *X* на единицу (как это было бы в языке высокого уровня). Это необходимо понимать как адрес, превышающий на единицу адрес переменной *X*. Выражение *X*+1 является примером адресного выражения.

2.5.3. Директива DB (Define Byte — определить байт)

В простейшем варианте директива DB имеет формат:

Имя **db** ***НачЗначение***

Операнд *НачЗначение* может быть в общем случае выражением, а в частном случае именем другой переменной (определенной ранее), константой, символом и знаком "?".

По этой директиве компилятор выделяет один байт памяти и закрепляет за ним указанное имя. Практически компилятор вводит псевдоним адреса выделенного байта, удобный для программиста. Кроме того, компилятор хранит информацию о том, что любой двоичный код, который будет находиться по этому адресу, будет представлять собой однобайтовую переменную.

Далее компилятор вычисляет выражение, стоящее в правой части, и его значением инициализирует описанную переменную. Если вместо операнда стоит знак "?", то инициализация не производится и значение переменной непредсказуемо.

Примеры:

```
p   db  1        ;Начальное значение p=1
q   db  2        ;Начальное значение q=2
r   db  ?        ;r не инициализировано
s   db  3+2      ;Начальное значение s=5
t   db  'Ю'      ;В байте t ASCII-код буквы Ю
```

Массив определяется либо как директива DB с несколькими операндами, либо как несколько директив db с именем только у первой директивы. В следующих двух примерах определяется один и тот же массив A с тремя элементами:

```
A       db      7, ?, 'z'               A       db      7
                                                db      ?
                                                db      'z'
```

Строка символов также рассматривается как массив из символов, но для нее имеется укороченная форма записи инициализации. В следующих двух примерах определяется одна и та же строка S с начальным значением "Наука умеет много гитик":

```
S  db 'Н','а','ука ','умеет ','много ','гитик'
S  db 'Наука умеет много гитик'
```

2.5.4. Директивы DW и DD (Define Word, Define Double Word)

Директивы DW и DD действуют практически так же, как директива DB (естественно, с учетом того, что определяют не байт, а слово или двойное слово). Соответственно расширяются диапазоны целых чисел, которые могут быть использованы для инициализации.

Не следует забывать, что байтовые части слов и двойных слов записываются в памяти в "перевернутом" виде (младшие байты по младшим адресам), поэтому не следует применять эти директивы для определения строк (символы в строках будут "перетасованы").

Инициализатором директивы DW может быть имя переменной, определенной ранее (или, в более общем случае, адресное выражение). Например:

```
A       db      7               ;В байте A число 00000111b
B       dw      A               ;В слове B смещение байта A
```

Использовать в директиве DD в качестве инициализатора константное выражение не имеет смысла, поскольку любая операция в константном выражении выполняется по модулю 2^{16}. Инициализатором директивы DD может быть имя переменной, определенной ранее (или, в более общем случае, адресное выражение). Например:

```
A        db      7                 ;В байте A число 00000111b
C        dd      A                 ;В младшем слове C смещение
                           ;байта A, в старшем слове C сегмент байта A
```

Конструкция повторения и определение массивов в директивах DW и DD применяются точно так же, как и в директиве DB.

2.6. Сегментная структура программы и модели памяти

2.6.1. Программные сегменты

Область памяти, которая адресуется при помощи регистра CS, будем называть **сегментом команд**. Область памяти, которая адресуется при помощи регистра DS, будем называть **сегментом данных**. Область памяти, которая адресуется при помощи регистра SS, будем называть **сегментом стека**. Естественно, в реальном режиме размер каждого такого сегмента не может превышать 64 Кбайт. Не следует думать, что это обязательно отдельные области памяти, например, один и тот же сегмент может быть одновременно и сегментом команд, и сегментом данных, и сегментом стека.

Поскольку команды Ассемблера транслируются в машинные команды, которые должны быть размещены в памяти по некоторым адресам, по директивам определения данных данные также должны быть размещены по некоторым адресам, мы должны определить в программе соответствующие **программные** сегменты команд, данных и стека.

Часть программы, которая оформляется в виде программного сегмента, заключается в директивы-скобки:

Имя_сегмента **segment** *Операнды*

...

...

...

Имя_сегмента **ends**

Следует обратить особое внимание на то, что определение программного сегмента не содержит указания на то, какому сегменту в памяти (команд, данных, стека) он будет соответствовать (за исключением случая указания операнда STACK).

Все операнды директивы SEGMENT необязательны (и чаще всего отсутствуют, за исключением операнда STACK). Может быть указано до 5 операндов в любом порядке через пробел:

Доступ	Выравнивание	Тип	Разрядность	'Класс'
READONLY	BYTE	PUBLIC	**USE16**	'*Метка*'
	WORD	STACK	USE32	
	DWORD	COMMON		
	PARA	AT *Адрес*		
	PAGE	**PRIVATE**		

В таблице жирным шрифтом выделены параметры, принятые по умолчанию. Не вдаваясь в подробное описание операндов и их значений, укажем лишь, что параметр '*Класс*' является как бы расширением имени сегмента (могут быть сегменты с одинаковыми именами и разными классами, и с разными именами и одинаковыми классами), а параметр *Тип* вместе с параметром '*Класс*' дает возможность реализовать различные способы соответствия между программными сегментами и сегментами в памяти. В относительно простых программах необходимо лишь указывать параметр STACK в описании программного сегмента стека.

Подчеркнем еще раз, что определение сегмента никак не указывает, при помощи какого сегментного регистра он будет адресоваться. Для этого применяется директива ASSUME, имеющая формат:

assume ***СегмРегистр: ИмяСегм,   СегмРегистр: ИмяСегм, ...***

В качестве ***СегмРегистр*** могут фигурировать имена сегментных регистров CS, DS,SS, ES (а также FS и GS для старших процессоров). В качестве ***ИмяСегм*** могут применяться имена сегментов, которые в дальнейшем будут фигурировать в директивах SEGMENT или зарезервированное слово NOTHING.

Директива ASSUME действует вниз по программе от места, где оно стоит, до следующей директивы ASSUME, которая отменяет действие предыдущей директивы ASSUME в отношении перечисленных в последней директиве сегментных регистров. Слово NOTHING отменяет действие предыдущих директив в

отношении данного сегментного регистра, но не устанавливает никакой новой его связи.

Из сказанного следует, что первая директива ASSUME должна стоять в начале программы вне программных сегментов.

Следует особо подчеркнуть, что для работы с сегментами необходимо выполнить еще одну операцию: **загрузку значений сегментных регистров**. Ни директива ASSUME, ни директива SEGMENT этого не делают, это должен обеспечить программист в начале выполнения программы. Впрочем, сегментные регистры команд и стека загружаются автоматически (операционной системой) и фактически необходимо лишь позаботиться о загрузке сегментного регистра данных.

2.6.2. Директивы модели памяти и упрощенного задания программных сегментов

Директивы ASSUME в совокупности с директивами SEGMENT позволяют создавать сложные программы со множеством сегментов.

Однако можно поступить иначе: сначала задать модель памяти, т. е. задать принцип соответствия программных сегментов и сегментов в памяти, а затем определить программные сегменты при помощи упрощенных директив. Подчеркнем, что при применении упрощенных директив предварительно **необходимо** задать модель памяти, а при применении директив ASSUME и SEGMENT задавать модель памяти **не нужно**.

Модель памяти задается директивой:

.model ***Модель***, *Язык*, *Спецификация*

Обязательный операнд ***Модель*** может принимать следующие значения:
TINY — Команды, данные и стек расположены в одном сегменте размером до 64 Кбайт.
SMALL — Команды расположены в одном сегменте, а данные и стек в другом сегменте.
COMPACT — Может быть несколько сегментов данных.
MEDIUM — Может быть несколько сегментов команд.
LARGE — Может быть несколько сегментов команд и данных.

HUGE — То же, что LARGE. Используется для совместимости с языками высокого уровня.
FLAT — Бессегментная организация памяти.
Для программ на Ассемблере (без взаимодействия с языками высокого уровня) практически используются модели TINY и SMALL. Модель COMPACT может применяться при обработке больших массивов данных.

Необязательный операнд *Язык* может принимать следующие значения: C, PASCAL, BASIC, FORTRAN. Он реализует некоторые умолчания и договоренности, принятые в языках высокого уровня.

Необязательный операнд *Спецификация* может принимать следующие значения: NEARSTACK (по умолчанию), FARSTACK (стек и данные в разных сегментах).

При применении директивы .MODEL становятся доступными следующие стандартные имена:
@CODE - - адрес сегмента команд,
@DATA - адрес сегмента данных типа NEAR,
@FARDATA - адрес сегмента данных типа FAR,
@FARDATA? — адрес сегмента неинициализированных данных,
@CURSEG — адрес текущего сегмента
@STACK — адрес сегмента стека

После того, как модель памяти определена, можно определить сегменты стека, данных и команд с помощью директив упрощенной сегментации, которые заменяют директивы ASSUME, SEGMENT и ENDS.

Директива упрощенного описания сегмента стека имеет вид
.stack *Размер*

Необязательный операнд *Размер* указывает максимальный размер стека в байтах. При отсутствии операнда размер стека полагается равным 1 Кбайт. Размер стека не должен быть меньше 128 байт плюс фактически используемая стековая память. Эту директиву нельзя применять при написании процедуры, встраиваемой в программу на языке высокого уровня (там стек организует сам компилятор языка). Сегментный регистр стека загружается автоматически.

Директива упрощенного описания сегмента данных имеет вид
.data

Директива определяет сегмент данных и связывает с ним сегментный регистр данных. Директива действует до следующей директивы упрощенной сегментации или до конца программы. В программе может быть несколько участков, озаглавленных этой директивой. При компиляции они будут объединены в один программный сегмент. Сегментный регистр данных необходимо загрузить значением @DATA (не непосредственно, а через некоторый РОН).

Директива упрощенного описания сегмента неинициализированных данных имеет вид **.data?**

Директива упрощенного описания сегмента команд имеет вид
.code *Имя*

Необязательный параметр *Имя* имеет смысл только для моделей памяти MEDIUM и выше (программные сегменты с разными именами будут объединены в разные сегменты). Директива действует до следующей директивы упрощенной сегментации или до конца программы. В программе может быть несколько участков, озаглавленных этой директивой. При компиляции они будут объединены в один программный сегмент. Сегментный регистр команд загружается автоматически.

С учетом всего сказанного структура простой программы принимает вид, приведенный в разделе 2.3 или в разделе 2.4. Если еще учесть то, что будет сказано о макроопределениях и процедурах, можно написать оболочку для получения исполняемого EXE-файла (файл EXESHELL.ASM) в виде

```
; СЮДА ВСТАВИТЬ ВСЕ МАКРООПРЕДЕЛЕНИЯ
        .model  small
        .stack                  ; Макс. объем стека 1 Кбайт
        .data
; СЮДА ВСТАВИТЬ ОПИСАНИЕ ВСЕХ ДАННЫХ
        .code
; СЮДА ВСТАВИТЬ ОПИСАНИЯ ВСЕХ ПРОЦЕДУР
Entry:  mov AX,@data
        mov   DS,AX
; СЮДА ВСТАВИТЬ КОМАНДЫ ГЛАВНОЙ ПРОГРАММЫ
        mov   AH,4Ch
```

```
        int    21h
        end    Entry
```

или, пользуясь включаемыми файлами, чтобы каждый раз не редактировать файл EXESHELL.ASM:

```
include  my_macros.asm                   ;Макроопределения
.386              ;Можно применять команды процессора 80386
.model small
.stack                            ; Макс. объем стека 1 Кбайт
.data
include  my_data.asm                     ;Определения данных
.code
include  my_procs.asm                     ;Описания процедур
Entry: mov    AX,@data
       mov    DS,AX
include  my_main.asm                  ;Нестандартные команды
       mov    AH,4Ch
       int    21h
       end    Entry
```

В файл с макроопределениями можно включить также некоторые из директив EQU, другие директивы EQU могут появляться в других частях программы по мере надобности, надо лишь следить, чтобы в результирующей программе каждая директива EQU предшествовала применению введенного с ее помощью псевдонима.

Вообще, при программировании при помощи включаемых файлов всегда необходимо мысленно представить, что включаемые файлы уже вписаны на свои места, и проверить, получается ли при этом *правильная* программа на Ассемблере.

Необходимо сделать следующее замечание: данные **могут** определяться в сегменте команд, и это часто делается. Однако при этом необходимо, во-первых, следить, чтобы на них случайно не попало управление, и, во-вторых, **явно** (при помощи префикса) адресовать их при помощи сегментного регистра команд.

По умолчанию сегменты загружаются в память (при выполнении программы) в том порядке, в котором они описаны в программе, причем если несколько программных сегментов объединяются в один, то порядок определяется по первому из объе-

диняемых программных сегментов. Этот порядок загрузки при желании можно изменить директивами .ALPHA и .DOSSEGM.

2.6.3. Директива изменения счетчика размещения ORG

Проводя трансляцию очередного программного сегмента и присваивая каждому предложению сегмента последовательно смещение относительно начала сегмента, компилятор ведет счетчик адресов (точнее, смещений относительно начала сегмента) предложений текущего сегмента. Значение этого счетчика в некотором предложении Ассемблера задается стандартной переменной $. Пример:

```
        mov     AX,$
```

По этой команде в AX загрузится смещение команды, в которую будет оттранслирована данная строка.

Значения этого счетчика можно изменять при помощи директивы:

org ***Выражение***

Эта директива означает, что команда, соответствующая следующей строке ассемблерной программы, будет размещена со смещением ***Выражение*** относительно начала текущего сегмента, например:

```
org     100h
xor     AX,AX    ;Эта команда будет размещена со смещением
                           ;100h относительно начала сегмента
org     $+13                            ; Пропускаем 13 байт
xor     BX,BX    ;Эта команда будет размещена со смещением
                           ;10Dh относительно начала сегмента
```

2.6.4. Структура исполняемых файлов. Исполняемые файлы EXE и COM

Исполняемые файлы, получающиеся после компиляции и компоновки, могут быть исполнены только в конкретной операционной системе. Операционная система выделяет память для сегментов программы, загружает саму программу и ее данные, настраивает сегментные регистры команд и стека и, наконец, передает управление на точку входа программы, настраивая регистры CS и IP на первую выполняемую команду программы. После выполнения программы управление, как правило, пере-

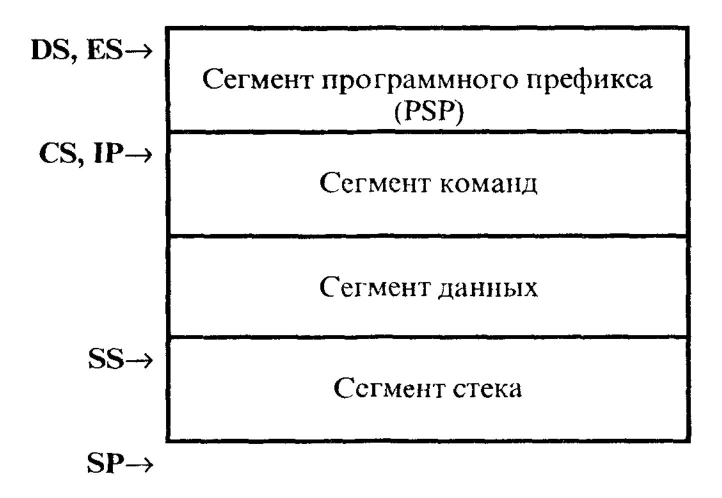
дается обратно в операционную систему (в DOS управление передается на программу COMMAND.COM, являющуюся частью операционной системы).

Если ограничиться операционной системой DOS, то она поддерживает два формата исполняемых файлов – EXE и COM. Операционная система распознает формат исполняемого файла не по расширению имени файла, а по его длине и по наличию в двух первых байтах EXE-файла символов MZ.

Следует отметить, что COM-формат исполняемого файла является устаревшим (исторически сохраняющимся с тех времен, когда объем оперативной памяти компьютера был всего 64 Кбайт). Никаких ощутимых преимуществ перед EXE-форматом COM-формат не имеет, а недостатки имеет существенные, поэтому с большой степенью достоверности можно предположить, что COM-формат исполняемых файлов обречен на вымирание.

Ассемблер позволяет получать исполняемые файлы, как в EXE, так и в COM-формате. Все, что говорилось ранее, относится к получению файлов в EXE-формате.

Рассмотрим структуру EXE-файла и его образ в памяти при загрузке на выполнение. EXE-файл содержит заголовок (relocation table) размером 512 байт, в котором содержится вся информация для операционной системы для загрузки сегментов в память и настройки сегментных регистров. При загрузке EXE-файла на выполнение операционная система создает в памяти определенное в программе число сегментов и еще один сегмент с фиксированным размером 256 байт, в который заносятся все необходимые данные выполняемой программы. Этот сегмент называется сегментом программного префикса (PSP – Program Segment Prefix). Например, если в программе определены отдельные сегменты команд, данных и стека именно в такой последовательности, то образ EXE-программы в памяти и исходные настройки сегментных регистров и регистров IP и SP можно изобразить следующим образом:



Сегментные регистры DS и ES первоначально настраиваются на сегмент программного префикса для того, чтобы, запомнив их значения, при необходимости к нему адресоваться. Именно поэтому возникает необходимость перенастройки сегментного регистра данных в самых первых выполняемых командах программы.

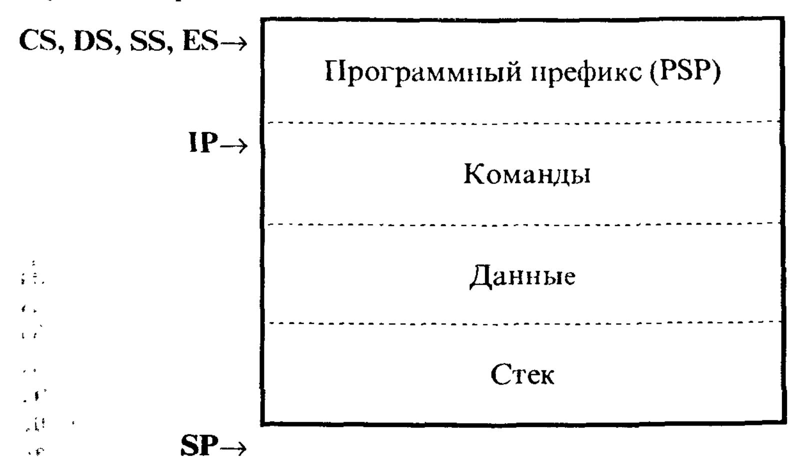
COM-программы устроены несколько иначе. Вся программа, подготавливаемая для COM-формата, должна иметь один сегмент, в котором размещаются и команды, и данные, и стек. Иными словами, используется модель памяти TINY. Отсюда сразу следует, что размер COM-программы в памяти не может превосходить 64 Кбайт. Другой особенностью COM-программы является то, что она вызывается операционной системой как процедура, поэтому нет необходимости завершать ее прерыванием 21*h*-4C*h*, команда возврата из процедуры RET (см. ниже в разделе "Процедуры") корректно завершит программу. В COM-программе нет необходимости в настройке сегментного регистра данных, поскольку сегмент всего один. Далее, в этом единственном сегменте необходимо предусмотреть 256 свободных байт вначале для программного префикса PSP. Это можно сделать, вставив директиву **org 100*h*** (это стандартное начало любой COM-программы). С учетом всего сказанного программа, предназначенная для получения COM-файла, примет вид (файл можно назвать COMSHELL.ASM):

```
        include   my_macros.asm      ; Макроопределения
        .model tiny
        .code
        org     100h
Entry:  include   my_main.asm    ;Нестандартные команды
        ret
        include   my_data.asm       ;Определения данных
        include   my_procs.asm      ;Описания процедур
        end     Entry
```

При программировании при помощи включаемых файлов всегда необходимо мысленно представить, что включаемые файлы уже вписаны на свои места, и проверить, получается ли при этом *правильная* программа на Ассемблере.

Далее, эту программу необходимо скомпилировать и скомпоновать с опцией **t** (опция для получения COM-файла). Полученный на диске исполняемый COM-файл будет приблизительно на 512 байт короче соответствующего EXE-файла (на диске) за счет отсутствия заголовка (relocation table).

В отличие от EXE-файла, код которого на диске сильно отличается от его образа в памяти при загрузке на выполнение, образ COM-файла в памяти является точной копией его кода на диске. Образ COM-программы в памяти и исходные настройки сегментных регистров и регистров IP и SP можно изобразить следующим образом:



Следует отметить, что в PSP со смещением 81*h* записывается командная строка программы, а в байт со смещением 80*h* — ее фактическая длина. (Поскольку размер PSP равен 100*h*, длина командной строки не может превосходить 100*h*-82*h*=7E*h*=126.) Это дает возможность проанализировать командную строку внутри программы и извлечь из нее параметры, передаваемые программе извне.

2.7. Команды Ассемблера

2.7.1. Общие сведения о командах

Напомним, что командами называются такие предложения Ассемблера, которые при компиляции непосредственно порождают машинные команды для процессора.

Каждая команда записывается в отдельной строке и имеет один из следующих форматов:

Метка:	***Мнемокод***		*;Комментарий*
Метка:	***Мнемокод***	***Операнд***	*;Комментарий*
Метка:	***Мнемокод***	***Оп1, Оп2***	*;Комментарий*

Поля метки, мнемокода, операндов и комментариев отделяются друг от друга некоторым числом пробелов или знаков табуляции. Операнды отделяются друг от друга запятой.

Строка может состоять из одной метки или из одного комментария.

Комментарии являются необязательной частью команд и в дальнейшем мы не будем их упоминать.

Метка также является необязательной частью команды, и в дальнейшем мы также не будем их упоминать там, где это не существенно, имея в виду, что любая команда может быть снабжена меткой.

В Ассемблере метки имеют более глубокий смысл, чем в языках высокого уровня. Чтобы понять смысл метки, необходимо иметь в виду, что компилятор транслирует команды Ассемблера в машинные команды и размещает их в памяти на свободном месте в сегменте команд. Таким образом, каждая команда (точнее, первый байт соответствующей машинной команды) имеет свое смещение относительно начала сегмента команд.

Если в команде Ассемблера указана метка, она является символическим именем этого смещения. Считается, что **значением метки** является смещение помеченной команды.

Итак, команда может иметь два операнда, иметь один операнд и не иметь его вовсе. Однако следует иметь в виду, что очень часто команды могут иметь некоторые операнды неявно, в виде подразумеваемых значений некоторых регистров (например, команда умножения MUL имеет лишь один явный операнд).

Команд в Ассемблере много, поэтому их следует разбить на функциональные группы:

- команды пересылки и преобразования,
- арифметические команды,
- команды передачи управления,
- команды работы со стеком,
- логические команды,
- команды сдвига,
- команды работы со строками,
- команды ввода/вывода и прерывания,
- другие команды.

По мере совершенствования процессора перечень команд все время растет. Имеется базовый набор команд, применимых для любого процессора 80x86. Чтобы его расширить, необходимо применить соответствующую директиву расширения (например, директиву **.486** для процессора Intel 486).

В командах с двумя явными операндами первый операнд называется **приемником**, а второй — **источником**, т. е. команда с двумя операндами имеет формат (опуская необязательные метку и комментарий):

Мнемокод Приемник, Источник

В качестве любого операнда, в принципе, могут быть указаны:

- константа (в более общем случае константное выражение), будем обозначать КВ,
- имя регистра (общего назначения или сегментного, кроме CS), будем обозначать РОН-8, РОН-16, СР,
- адрес ячейки памяти (указанный тем или иным способом), будем обозначать ПП — переменная в памяти.

В командах с двумя операндами оба операнда **не могут быть** одновременно адресами ячеек памяти!

2.7.2. Адресация

В тех случаях, когда операндом команды является имя регистра (не взятое в квадратные скобки!), особых вопросов не возникает.

Пример: `mov AX,BX`

Здесь содержимое регистра BX загружается в регистр AX

В тех случаях, когда операндом команды является константа, иногда появляется необходимость указать, какого типа константа имеется в виду. Например, не ясно, является ли константа 7 байтом, словом или двойным словом. Иногда компилятор сам может решить этот вопрос, исходя из вида другого операнда или самой команды. В других случаях необходимо явно указать тип константы при помощи конструкции ***Тип*** **PTR**, например:

BYTE PTR 7 — Константа 7 типа байт
WORD PTR 7 — Константа 7 типа слово
DWORD PTR 7 — Константа 7 типа двойное слово

Пример:

```
mov   AX,7     ;Константа 7 загружается в AX, в AX 0007h
mov   AH,7     ;Константа 7 загружается в AH, в AX 07??h
```

Гораздо сложнее обстоит дело, когда в качестве операнда тем или иным способом указывается адрес ячейки памяти. Способы указания ячейки памяти в качестве операнда называются **способами адресации**, или просто адресацией. Способы адресации определяются как синтаксисом Ассемблера, так и моделью процессора.

По мере совершенствования процессоров способы адресации усложняются (вводятся новые способы адресации). Имеется базовый перечень способов адресации, применимых для любого процессора 80x86. Чтобы его расширить, необходимо применить соответствующую директиву расширения (например, директиву **.486** для процессора Intel 486). Рассмотрим базовый перечень способов адресации.

Поскольку любой адрес состоит из сегмента и смещения, рассмотрим сначала вопрос об определении сегментной части адреса. В операнде при помоши префикса можно явно указать сегментный регистр, по которому определяется сегментная часть адреса, например:

DS: ***Указание_на_смещение*** — сегментная часть операнда в регистре данных

CS: ***Указание_на_смещение*** — сегментная часть операнда в регистре команд

SS: ***Указание_на_смещение*** — сегментная часть операнда в регистре стека

ES: ***Указание_на_смещение*** — сегментная часть операнда в дополнительном регистре

Однако чаще всего явно указывать сегментный регистр нет необходимости. Существуют правила умолчания, в соответствии с которыми определяется сегментная часть адреса. Эти правила умолчания действуют в зависимости от способа указания на смещение и будут приводиться при рассмотрении соответствующих способов адресации.

2.7.2.1. Прямая адресация

Прямой адресацией называется тот случай, когда смещение адреса ячейки памяти указывается в операнде непосредственно. При этом возможны 3 случая:

1. Операнд — имя переменной (ранее определенной). В этом случае сегментная часть по умолчанию определяется сегментным регистром данных.
2. Операнд — метка другой команды. В этом случае сегментная часть по умолчанию определяется сегментным регистром команд.
3. Операнд — константа. В этом случае сегментный регистр необходимо определить явно при помощи префикса.

Пример:

```
        .data
MyVar   db    7
MyStr   db    'Наука умеет много гитик'
        ...
        .code
        ...
```

```
MyLbl: mov   AX,MyVar        ;Загрузка значения переменной
                             ;MyVar в регистр AX
       mov   BX,MyVar+1      ;Загрузка значения следующего
                             ;байта после MyVar в BX
       lea   DX,MyStr        ;Загрузка адреса первого байта
                             ;MyStr в регистр DX
       ...
       jmp   MyLbl           ;Безусловный переход на
                             ;команду с меткой MyLbl
       ...
       jmp   CS:0000h        ;Безусловный переход на
                             ;первую команду сегмента команд
       ...
```

2.7.2.2. Косвенная адресация

Косвенной адресацией называется тот случай, когда в операнде указывается не сам адрес, а "место", где хранится этот адрес. Такими "местами" могут быть:

- регистры BX, SI и DI,
- регистр BP при работе со стеком,
- регистры EAX, EBX, ECX, EDX, EBP, ESI, EDI в режиме расширенной адресации для старших моделей процессоров.

В общем случае косвенной адресации в отдельных регистрах хранятся только "части" смещения, из этих "частей" по определенным правилам образуется так называемый **исполнительный адрес.**

В самом общем виде исполнительный адрес записывается по схеме (для младших процессоров):

DS:	***АдресноеВыражение*** +	**[BX]**	+	**[SI]**
CS:		**[BP]**		**[DI]**
SS:				
ES:				

Эту схему необходимо понимать следующим образом. Из вертикального столбца можно выбрать лишь один (любой) элемент. Все сложения производятся по модулю 2^{16}.

Любой элемент может отсутствовать. При этом, если отсутствует префикс сегментного регистра, вступает в действие правило умолчания, согласно которому при наличии регистра [BP] сег-

ментация проводится по регистру стека SS, при отсутствии регистра [BP] — по регистру данных DS.

Регистр [BP] применяется при работе со стеком, регистр [BX] — во всех остальных случаях.

Имена регистров-модификаторов обязательно должны стоять в квадратных скобках (в отдельных или в общей квадратной скобке для всего операнда).

В одном операнде не должны одновременно фигурировать регистры [BX] и [BP] или регистры [SI] и [DI], т. е. для модификации разрешены пары (BX,SI), (BX,DI), (BP,SI), (BP,DI) и запрещены пары (BX,BP), (SI,DI).

Частные случаи косвенной адресации имеют свои названия:

- Адресацию с использованием базовых регистров называют **адресацией по базе**.
- Адресацию с использованием индексных регистров называют **адресацией с индексированием**.
- Адресацию с применением константы называют **адресацией со сдвигом**.

Легко видеть, что прямая адресация является частным случаем косвенной, когда в операнде не применяются регистры-модификаторы.

Не разрешается заключать в квадратные скобки имена регистров, не являющихся регистрами модификаторами.

Следует отметить, что квадратные скобки имеют крайне неудачную алгебру: [x][y]=[x+y]=[y][x], причем любую часть выражения, не содержащую регистров, можно заключить в квадратные скобки и можно, наоборот, опустить квадратные скобки. Это ведет к тому, что один и тот же адрес можно записать многими различными способами, например: A[BX][DI], A[BX+DI], [A+BX+DI], A[BX]+[DI] и т. д. Иногда можно записать операнд в форме, напоминающей элемент массива в языках высокого уровня A[DI+1].

2.7.3. Команды пересылки и преобразования

Для описания команд введем следующие сокращения:

- КВ — константное выражение

- РОН-8 — регистр общего назначения (AH, AL, BH, BL, CH, CL, DH, DL)
- РОН-16 — регистр общего назначения (AX, BX, CX, DX, SI, DI, BP, SP)
- СР — сегментный регистр (SS, DS, CS, ES)
- РМ — регистр-модификатор (BX, BP, SI, DI), входят в число РОН-16
- ПП — переменная в памяти.

Все команды этого раздела не меняют значений флагов.

2.7.3.1. Команда пересылки MOV

Формат: **mov** ***Приемник, Источник***

Процессор: 8086

Источник: КВ, РОН, СР, ПП

Приемник: РОН, СР (кроме CS), ПП

Приемник и источник не могут быть одновременно переменными в памяти, не могут быть одновременно сегментными регистрами, источник должен быть РОН-16 при приемнике — сегментном регистре.

Оба операнда должны быть одного и того же размера: байт, слово, двойное слово. Никаких преобразований при этом не производится, за исключением удлинения коротких констант слева незначащими нулями. Если размер операндов не ясен из контекста, его необходимо уточнить явно с помощью конструкции ***Тип*** **PTR**.

Команда аналогична операторам присваивания языков высокого уровня: ***Приемник:=Источник;***

Примеры:

```
mov  AH,BH          ;Загрузка AH содержимым BH
mov  BX,AL          ;ОШИБКА: размерности регистров
                    ;не совпадают
mov  [BX],AL        ;Пересылка байта из AL в память по
                    ;адресу [DS]:[BX]
mov  ES:[BX],AL     ;Пересылка байта из AL в память по
```

```
                                   ;адресу [ES]:[BX]
mov  DS,DX                         ;Загрузка DS из DX
mov  CS,DX                         ;ОШИБКА: CS загружать нельзя
mov  DS,ES                         ;ОШИБКА: оба операнда — сегм. регистры
mov  ES,0B000h                     ;ОШИБКА: сегментный регистр нельзя
                                   ;загружать константой
mov  DS,@data                      ;ОШИБКА: сегментный регистр нельзя
                                   ;загружать напрямую
mov  BX,SI                         ;Загрузка BX содержимым SI
mov  [BX],[SI]                     ;ОШИБКА: оба операнда — в памяти
mov  [SI],3                        ;ОШИБКА: компилятор не в состоянии
                                   ;определить тип операндов
mov  byte ptr[SI],3                ;Число 3 загружается в байт
                                   ;по адресу[DS]:[SI]
mov  [SI],byte ptr 3               ;Эквивалентно предыдущему
mov  AX,1                          ;Загрузка 0001 в AX
mov  AH,280                        ;ОШИБКА: конст. 280 больше байта
mov  A[DI],0                       ;Обнуление (n+1)-го элемента массива
                                   ;байт A, если в DI имеется число n
mov  A[DI-1],0                     ;Обнуление n-го элемента массива
                                   ;байт A, если в DI имеется число n
```

Следует отметить, что одной команде **mov** в Ассемблере соответствует много команд машинного кода (более 20).

2.7.3.2. Команды условной пересылки CMOVxx

Формат: **cmovXXX** ***Приемник, Источник***

Процессор: Pentium II

Источник: РОН, ПП

Приемник: РОН

Операнды должны быть одинаковой длины.

Команды имеют постфиксы XXX, определяющие условие, при котором происходит пересылка. Постфиксы, соответствующие

им условия и соответствующие значения флагов будут рассмотрены для команды Jxxx (см. с. 74). Для установки надлежащих значений флагов непосредственно перед командой необходимо выполнить команду CMP с теми же операндами.

2.7.3.3. Команда обмена операндов XCHG (Exchange)

Формат: **xchg** ***Приемник, Источник***

Процессор: 8086

Источник: РОН, ПП

Приемник: РОН, ПП

Операнды должны быть одинаковой длины и не могут одновременно быть переменными в памяти.

Примеры:

```
xchg    BL,BL        ;Не делает ничего
xchg    BH,BL        ;Обмен байт в регистре BX
xchg    AX,BX        ;На С: tmp=AX; AX=BX; BX=tmp;
```

2.7.3.4. Команда обмена байт в 32-регистре BSWAP

Формат: **bswap** ***32_регистр***

Процессор: 486

Операнд: 32-разрядный регистр

Меняются местами младший и старший байты, а также меняются местами средние байты.

Пример:

```
mov   EAX,12345678h
bswap EAX            ;В регистре EAX 78563412h
```

2.7.3.5. Команда конвертирования байта в слово CBW (Convert Byte to Word)

Формат: **cbw**

Процессор: 8086

Операндов нет

Расширяет байт в AL до слова в AX. Можно трактовать, как заполнение битов AH значениями, равными значению старшего бита AL.

Пример:

```
mov    AL,-11                  ;AL=245=0F5h
cbw                            ;AX=0FFF5h=65525=-11
```

2.7.3.6. Команды конвертирования слова в двойное слово CWD и CWDE (Convert Word)

Формат: **cwd** или **cwde**

Процессор: 8086 или 386

Операндов нет

Команда CWD расширяет слово в AX до двойного слова, младшая часть которого находится в AX, а старшая часть — в DX. Команда CWDE расширяет слово в AX до двойного слова в регистре EAX. Можно трактовать, как заполнение битов старшей части значениями, равными значению старшего бита младшей части.

2.7.3.7. Команда перекодировки в соответствии с таблицей XLAT (Translation)

Формат: **xlat**

Процессор: 8086

Операндов нет

Предполагается, что в сегменте данных имеется массив байт (строка символов) длины не более 256 байт. Перед применением команды в регистр BX необходимо загрузить смещение начала этого массива, а в регистр AL загрузить некоторое число в диапазоне от 0 до числа элементов этого массива (например, 143), тогда после выполнения этой команды в регистре AL окажется 143-й элемент этого массива. Практически происходит перекодировка ASCII-кода символа, стоящего на *n*-м месте в таблице ASCII, на код символа, стоящий на *n*-м месте в заданной таблице.

2.7.3.8. Команда загрузки исполнительного адреса LEA (Load Effective Address)

Формат: **lea** ***Приемник, Источник***

Процессор: 8086

Источник: ПП

Приемник: РОН-16

Вычисляется исполнительный адрес источника и загружается в регистр приемника. Важно понять ее отличие от команды Mov. LEA отличается от команды MOV тем, что по команде MOV в приемник загружается значение по исполнительному адресу, а по команде LEA — сам исполнительный адрес. Например, если определена переменная Msg, то эквивалентны следующие команды:

```
mov   DX,offset Msg
```

и

```
lea   DX,Msg
```

2.7.4. Команды двоичной арифметики

2.7.4.1. Команды сложения, вычитания и сравнения ADD, SUB, ADC, SBB, INC, DEC, CMP

Операции выполняются одинаково для знаковых и для беззнаковых величин.

Сложение и вычитание байт выполняется по модулю 2^8, сложение и вычитание слов — по модулю 2^{16}.

Сложение по модулю 2^n означает, что если результат превысил 2^n, из него автоматически вычитается 2^n, т. е. фактически отбрасывается единица старшего $(n+1)$-го разряда. Вычитание по модулю 2^n означает, что если вычитаемое больше уменьшаемого, то к уменьшаемому заранее прибавляется 2^n, т. е. фактически производится "заем" единицы старшего $(n+1)$-го разряда, и только после этого производится вычитание.

Если при выполнении действия заем или отбрасывание единицы имели место, флаг переноса CF выставляется в 1, иначе он выставляется в 0.

Если при выполнении действия происходит перенос в старший (знаковый) бит, флаг переполнения OF выставляется в 1, иначе он выставляется в 0.

Если при выполнении действия результат равен нулю, флаг переноса ZF выставляется в 1, иначе он выставляется в 0.

Флаг знака результата SF выставляется равным значению старшего (знакового) бита результата.

Все команды действуют для процессора 8086.

Команды сложения, вычитания и сравнения ADD, SUB, CMP:

add ***Приемник,Источник*** ;Приемник:= Приемник + Источник
sub ***Приемник,Источник*** ;Приемник:= Приемник-Источник
cmp ***Операнд1, Операнд2*** ; Операнд1 – Операнд2

Источник: КВ, РОН, ПП

Приемник: РОН, ПП

Приемник и источник не могут быть одновременно переменными в памяти.

Оба операнда должны быть одного и того же размера: байт или слово.

Команда CMP действует абсолютно так же, как команда SUB, но результат никуда не записывается. Команда CMP служит для выставления нужных значений флагов.

Команды сложения с переносом и вычитания с заемом ADC и SBB:

adc ***Пр,Ист*** ;Приемник:= Приемник + Источник + [CF]
sbb ***Пр, Ист*** ;Приемник:= Приемник – Источник – [CF]

Действуют точно так же, как команды ADD и SUB, но с прибавлением или вычитанием старого значения флага CF. Применяются для организации сложения и вычитания "длинных" чисел.

Команды инкремента и декремента INC и DEC:

inc ***Операнд*** ; Операнд:= Операнд + 1
dec ***Операнд*** ; Операнд:= Операнд - 1

Операнд должен быть регистром, словом или байтом **в памяти.**

Эти команды не меняют флага CF.

2.7.4.2. Команды умножения и деления MUL, IMUL, DIV, IDIV

Команда умножения без знака: **mul** *Источник*

Команда умножения со знаком:**imul** *Источник*
Источник: РОН, ПП
Процессор: 8086

При перемножении байтов второй операнд должен находиться в AL, а результат записывается в AX.

При перемножении слов второй операнд должен находиться в AX, младшая часть результата записывается в AX, а старшая — в DX.

Итак, для результата всегда отводится вдвое больше места, чем для каждого операнда. Однако фактически часто результат может требовать столько же места, что и операнды, т. е. произведение байт может умещаться в байте, произведение слов — в слове. Чтобы различить эти случаи, используются флаги CF и OF (которые при умножении изменяются одинаково). Если произведение требует удвоенного формата, эти флаги выставляются в 1, если для произведения достаточно формата сомножителей, CF и OF выставляются в 0.

Для процессоров 186 и старше **только для перемножения слов** (и для результата в формате слова) введены следующие, более удобные, разновидности команд умножения:

Команда умножения без знака:
mul *Приемник, Источник1, Источник2*
Команда умножения со знаком:
imul *Приемник, Источник1, Источник2*
Приемник: РОН, ПП
Источник 1: РОН, ПП
Источник 2: РОН, ПП, КВ

При переполнении флаги CF и OF **выставляются в** 1, **иначе выставляются** в 0.

Эти команды можно записывать и в сокращенной форме, с двумя операндами

mul *Приемник, Источник*

что эквивалентно

mul *Приемник, Приемник, Источник*

Команда деления без знака: **div** *Делитель*
Команда деления со знаком: **idiv** *Делитель*

Для деления слова на байт делимое должно находиться в AX, а делитель должен быть байтом в памяти или байтовым регистром. При этом в AL получается частное (целое) от деления, а в AX — остаток от деления.

Для деления двойного слова на слово старшая часть делимого должна находиться в DX, младшая часть делимого должна находиться в AX, а делитель должен быть словом в памяти или РОН-16. При этом в AX получается целое частное от деления, а в DX — остаток от деления.

При делении возможны ошибки двух типов, при которых программа аварийно завершается:

- Деление на 0 (zerodivide): делитель равен нулю.
- Переполнение (divide overflow): частное не вмещается в отведенный формат.

2.7.5. Команды передачи управления и циклы

Уже говорилось, что программа, как правило, выполняется в естественной последовательности, т. е. в порядке написания ее команд сверху вниз. Фактически это происходит следующим образом: в регистр IP автоматически загружается смещение следующей команды, и процессор выбирает для выполнения команду по адресу CS:IP.

Поэтому прервать эту естественную последовательность выполнения можно, загрузив в регистр IP (и, может быть, в CS) некоторые другие значения. Однако сделать это впрямую (например, командами mov или lea) невозможно. Для этого существуют специальные команды передачи управления (или перехода).

Следует различать переходы **вперед и назад** по программе. Переходы назад нельзя рекомендовать, поскольку они значительно ухудшают понимаемость программы и усложняют ее логику. Переходы вперед порождают проблемы при компиляции, поскольку в строке перехода метка перехода еще не определена (требуется задавать двухпроходный режим компилятора).

Переходы бывают:

- короткими (**short**) — в пределах ±128 байт от команды перехода
- ближними (**near**) — в пределах одного программного сегмента
- дальними (**far**) — в другой программный сегмент
- межпрограммными — в другую программу в защищенном режиме.

Мы будем рассматривать только короткие и ближние переходы.

Переходы бывают **безусловными и условными**, т. е. происходящими только при выполнении какого-либо условия (тех или иных значений флагов).

Переходы могут не сохранять никакой информации для возврата в точку перехода или сохранять такую информацию (в этом случае переход называется **переходом с возвратом или вызовом подпрограммы**).

2.7.5.1. Безусловный переход JMP

Формат	**jmp**	***Операнд***
Операнд:	метка, РОН, ПП	
Процессор:	8086	

В общем случае формируется машинная команда ближнего (near) перехода, по которой фактически к значению регистра IP прибавляется (или вычитается) двухбайтовое смещение относительно команды перехода

Однако в перечисленных ниже случаях формируется машинная команда короткого (short) перехода, по которой фактически к значению регистра IP прибавляется (или вычитается) однобайтное смещение относительно команды перехода:

- Компилятор сам в состоянии выяснить, что переход происходит в пределах ±128 байт.

- В команде указан оператор short: **jmp short** ***Операнд***
- В качестве операнда задан однобайтный РОН или байт памяти.

Команда короткого перехода выполняется быстрее и занимает меньше места в памяти, чем команда ближнего перехода.

Так же, как и адресация, переход может быть прямой и косвенный. **Прямым переходом** считается переход на метку.

Кроме обычных меток, в Ассемблере предусмотрены специальные стандартные метки @@, которых в программе может быть любое число. По команде **jmp @B** происходит переход на ближайшую метку @@ вверх по программе, по команде **jmp @F** происходит переход на ближайшую метку @@ вниз по программе.

Косвенным переходом считается переход по адресу, который хранится в некоторой ячейке памяти или РОН. Косвенные переходы дают возможность осуществлять переходы динамически в разные места в зависимости от ситуации, складывающейся при выполнении программы.

2.7.5.2. Условные переходы Jxxx

Условные переходы имеют формат **jxxx** ***Метка***

Все они осуществляют **короткий переход** при некотором условии, определяемом постфиксом мнемокода **xxx**. Все эти условия определяются состояниями флагов (за исключением команды JCXZ, условие перехода которой определяется состоянием регистра CX).

Если необходимо осуществить условный **близкий** (не короткий) переход, приходится комбинировать условный и безусловный переходы.

Вообще говоря, условие перехода определяется только состояниями флагов, однако чтобы не заниматься анализом значений флагов, их можно выставить при помощи примененной непосредственно перед условным переходом команды

cmp ***Операнд1, Операнд2***

Тогда комбинации флагов приобретают определенный математический смысл, указанный в следующей таблице:

Постфикс	English	Переход после **cmp x,y** , если:	Условия на флаги
	ЛЮБЫЕ ЧИСЛА		
E	Equal	x=y	ZF=1
NE	Not Equal	x≠y	ZF=0
	ЧИСЛА СО ЗНАКОМ		
L / NGE	Less / Not Greater Equal	x<y	SF≠OF
LE / NG	Less Equal / Not Greater	x≤y	SF≠OF или ZF=1
G / NLE	Greater / Not Less Equal	x>y	SF=OF и ZF=0
GE / NL	Greater Equal / Not Less	x≥y	SF=OF
	ЧИСЛА БЕЗ ЗНАКА		
B / NAE	Below / Not Above Equal	x<y	CF=1
BE / NA	Below Equal / Not Above	x≤y	CF=1 или ZF=1
A / NBE	Above / Not Below Equal	x>y	CF=0 и ZF=0
AE / NB	Above Equal / Not Below	x≥y	CF=0
	ПОСЛЕ ДРУГИХ КОМАНД		
Z / NZ	Zero / Not Zero(флаг нуля)		ZF= 1 / 0
S / NS	Sign / Not Sign (флаг знака)		SF= 1 / 0
C / NC	Carry / Not Carry (флаг заема)		CF= 1 / 0
O / NO	Overflow / Not Overflow (флаг переполнения)		OF= 1 / 0
P / NP	Parity / Not Parity (флаг четности)		PF= 1 / 0
	ПЕРЕХОД ПО НУЛЮ CX		
CXZ	CX equal to Zero	CX=0	--

Перечисленные в таблице условия действуют и для команд **CMOVxxx** с аналогичными постфиксами.

2.7.5.3. Команды циклов LOOP, LOOPZ, LOOPNZ

Команда LOOP имеет формат:

```
        mov  CX, Источник       ;Загр. в CX числа повторений
Метка: ...
        ...                      ; Тело цикла
        ...
        loop Метка
```

По команде LOOP происходит уменьшение значения CX на единицу и, если CX≠0, возврат на помеченную первую команду тела цикла. Перед циклом в CX должно быть загружено положительное число повторений цикла.

Команда LOOP осуществляет только **короткий** переход

Команды LOOPZ и LOOPNZ действуют точно так же, как команда LOOP, но к условию повторения цикла добавляется условие ZF=1 для LOOPZ и ZF=0 для LOOPNZ.

Пример (заполнение массива квадратами натуральных чисел):

```
n       equ   100
        .data
x       dw    n dup(?)       ;Массив из n=100 слов xi (i=0,..., 99)
        .code
Entr:   mov   AX,@data
        mov   DS,AX
        mov   DI,0                  ;Нач. значение индекса i=0
        mov   CX,n                  ;Число повторений
Cycl:   movAX,DI                    ;Индекс массива i загр. в AX
        mul   AX                    ;Квадрат индекса i2
        mov   x[DI],AX       ;Элемент равен квадрату индекса
        inc   DI      ;Переход к следующему значению индекса
        loop  Cycl                  ;Повторить n раз
        mov   AH,4Ch         ;Подготовка к возврату в DOS
        int   21h
        end   Entr
```

2.7.5.4. Команды вызова процедуры CALL и возврата из процедуры RET

Формат вызова: **call** ***Имя_Процедуры***

Формат возврата: **ret**

или **ret** ***Четное_число***

По команде вызова в стек записывается адрес следующей команды и осуществляется переход (ближний или дальний в зависимости от описания процедуры) в тело процедуры.

По команде возврата осуществляется переход на команду, адрес которой определяется верхним элементом стека. При указании в качестве параметра целого числа, после возврата в вызывающую программу регистр указателя стека увеличивается на это число, т. е. из стека удаляется ***Четное_число***/2 элементов.

2.7.6. Команды работы со стеком

Стеком называется особая программная структура, предусмотренная в отдельной области памяти (в сегменте стека), с которой можно работать двумя способами: как с обычной памятью и специальным образом при помощи специальных команд.

В основном стек используется для временного хранения некоторых величин, которым нецелесообразно давать индивидуальные имена, а также для автоматического сохранения некоторых величин (например, адреса возврата из процедуры).

Стек — это последовательность **слов** в памяти, организованная по принципу "последним пришел — первым ушел" (LIFO — Last In First Out). Стек можно представлять себе как лежащую на столе стопку листов бумаги (каждый размером в слово), с которой возможны две операции: положить на стопку новый лист бумаги и снять со стопки верхний лист бумаги. При этом необходимо иметь в виду, что имеется принципиальная возможность ознакомиться с содержимым листа где-то внутри стопки (не снимая верхние листы), но в этом случае мы будем вынуждены действовать не с помощью специфических стековых команд, а по общим правилам работы с памятью.

Напомним, что сегмент стека определяется двумя способами:

Имя	**segment stack**	
	db	***Размер*** **dup(?)**
	ends	***Имя***

или сокращенно: **.stack** ***Размер***

Следующей должна идти директива определения другого сегмента.

Если ***Размер*** отсутствует, он полагается равным 1 Кбайт.

Размер стека не должен быть меньше 128 байт и не должен превосходить 64 Кбайт. Сегмент стека необходимо определять даже в программе, явно не использующей стек, поскольку в любой программе применяются прерывания, а они используют стек.

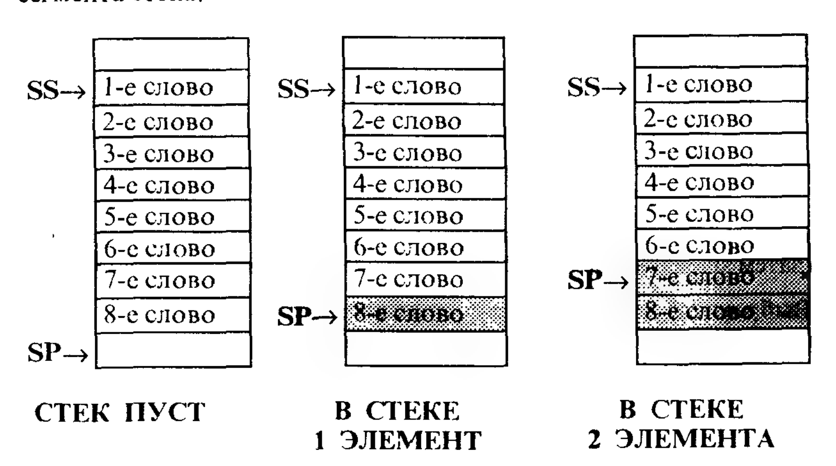
Сегментный регистр стека SS загружать не надо — он загружается автоматически при запуске программы.

Работа со стеком осуществляется при помощи регистра указателя стека SP, который указывает на вершину стека.

Чтобы графически изобразить стек, допустим, что его сегмент определен директивой **.stack 16** (хотя это и неверно — размер должен быть не меньше 128 байт). Это означает, что в стеке максимально может быть 8 слов.

В начале действия программы стек пуст. Его сегментный регистр SS автоматически настраивается на начало сегмента стека, а в регистр указателя стека SP автоматически заносится максимально возможный размер стека (в данном случае 16). Это можно понимать как занесение в регистр SP смещения первой ячейки памяти, лежащей ниже за пределами стека, и как отражение того факта, что стек пуст.

Вталкиваем в стек первое слово (при помощи команды PUSH). При этом значение SP автоматически уменьшается на 2, т. е. в SP записывается новое смещение вершины стека относительно сегмента стека.



Из схемы видно, что стек растет снизу вверх (т. е. в сторону убывания адресов). Соответственно сокращаться он будет сверху вниз (т. е. в сторону возрастания адресов).

Понимание того, как стек расположен в памяти, нужно только для того, чтобы работать со стеком, как с обычной памятью. Если мы собираемся работать со стеком при помощи специфических стековых команд, мы можем забыть о его расположении в памяти, а помнить только принцип "последним пришел — первым ушел". В соответствии с этим принципом каждой команде вталкивания в стек PUSH должна соответствовать своя команда выталкивания из стека POP, т. е. программа должна быть сбалансирована по командам PUSH и POP.

2.7.6.1. Команды вталкивания в стек PUSH и выталкивания из стека POP

Формат:	**push**	***Источник***
	pop	***Приемник***
Процессор:	8086	
Источник:	РОН-16, СР, ПП, КВ	
Приемник:	РОН-16, СР (кроме CS), ПП	

По команде PUSH значение регистра SP уменьшается на 2 (по модулю 2^{16}) и затем в новую вершину стека вталкивается содержимое источника (обязательно слово). Начиная с процессора 186, в качестве источника можно использовать константное выражение. В частном случае, по команде PUSH SP процессоры до 286 включительно записывают новое значение SP, процессоры начиная с 386 — старое значение SP.

По команде POP в приемник загружается слово из вершины стека и затем значение регистра SP увеличивается на 2 (по модулю 2^{16}).

"Выбросить" из стека *n* ненужных слов можно просто увеличением SP на 2*n*, т. е. можно выполнить команду:

```
add     SP,2*n
```

(можно, конечно, *n* раз применить команду POP).

2.7.6.2. Команды сохранения в стеке регистра флагов PUSHF и POPF

Формат: **pushf** или **popf**
Процессор: 8086

Команда PUSHF вталкивает в стек содержимое регистра флагов Она, как и все остальные команды работы со стеком (кроме POPF), не меняет флаги.

Команда POPF выталкивает из стека очередное слово и загружает его в регистр флагов. При этом она, естественно, устанавливает заново все флаги.

Отметим, что пара PUSHF-POP дает единственную возможность работать с содержимым FLAGS как со словом.

2.7.6.3. Команды сохранения в стеке регистров общего назначения PUSHA и POPA

Формат: **pusha** или **popa**
Процессор: 80186

Команда PUSHA вталкивает в стек содержимое всех РОН в следующей последовательности:

AX, BX, CX, DX, SP (старое значение), BP, SI, DI.

Команда POPA выталкивает из стека очередные 8 слов и загружает их в РОН в обратной последовательности (для SP берется новое значение).

Для процессоров 386 и выше имеются 32-битные аналоги этих команд PUSHAD и POPAD для сохранения в стеке расширенных регистров.

2.7.6.4. Работа со стеком, как с обычной памятью

Работа с памятью стека как с обычной памятью (например, при необходимости получить доступ к элементам внутри стека) не имеет особенностей, за исключением того, что при этом выгодней использовать специальный регистр-модификатор стека BP. Выгода заключается в том, что при модификации адресов при помощи BP сегментация адресов по умолчанию будет производиться по сегментному регистру стека SS (а не как обычно по DS).

Например, для обращения к некоторой ячейке внутри стека (не обязательно к слову -- может быть и байт и двойное слово) можно загрузить в BP значение регистра SP (т. е. выполнить команду **mov BP, SP**) и затем обращаться к *n*-му байту стека при помощи адресного выражения [BP+n-1].

2.7.7. Логические команды

Логические команды выполняются побитно, т. е. значение *n*-го бита результата определяется только значениями *n*-х битов операндов.

Для команд с двумя операндами оба операнда могут быть либо байтами, либо словами (но не байтом и словом).

Для команд с двумя операндами первый операнд (Приемник) может быть либо РОН, либо переменной в памяти, второй операнд (Источник) может быть либо РОН, либо переменной в памяти, либо константным выражением. Оба операнда, как обычно, не могут одновременно быть переменными в памяти.

Логические команды, вообще говоря, изменяют значения флагов, однако самым важным является флаг нуля ZF.

При необходимости создания байтовых логических величин, аналогичных константам TRUE и FALSE в Паскале, можно использовать следующие константы:

```
        true  equ    11111111b    ; =0FFh
        false equ    00000000b    ; =0h
```

Легко проверить, что для таких констант алгебра логики будет выполняться.

2.7.7.1. Команды логического умножения (конъюнкции) AND и TEST

Формат: **and** ***Приемник, Источник***

test ***Операнд1, Операнд2***

n-й бит результата равен 1 только тогда, когда оба *n*-х бита операндов равны 1, в противном случае 0.

Команда TEST отличается от команды AND только тем, что результат никуда не записывается.

Команду AND можно применять для выборочного обнуления отдельных битов, например:

```
and   AL,00111111b        ;Обнуле́ние двух старших битов AL
```

Команду TEST можно применять для выборочной проверки равенства нулю отдельных битов, например:

```
test  AL,10000000b        ;Проверка равенства нулю старшего
                          ;бита AL (по значению ZF)
```

2.7.7.2. Команда логического сложения (дизъюнкции) OR

Формат: **or** ***Приемник, Источник***

n-й бит результата равен 1 тогда, когда хотя бы один из *n*-х битов операндов равен 1, в противном случае 0.

Команду OR можно применять для выборочной установки в 1 отдельных битов, например:

```
or    AL,11000000b        ;Установка в 1 двух старших битов AL
```

2.7.7.3. Команда исключающего логического "ИЛИ" XOR

Формат: **xor** ***Приемник, Источник***

n-й бит результата равен 1 тогда, когда только один из *n*-х битов операндов равен 1, в противном случае 0.

Команду XOR можно применять для выборочного переключения отдельных битов или для обнуления, например:

```
xor   AL,11000000b        ;Переключение двух старших битов AL
xor   AL,AL               ;Наилучший способ обнуления AL
```

2.7.7.4. Команда логического отрицания NOT

Формат: **not** ***Приемник***

Каждый бит приемника инвертируется. Флаги **не изменяются.**

2.7.8. Команды сдвига

Все команды сдвига имеют формат

Мнемокод ***Приемник, Счетчик***

Приемником может быть РОН или переменная в памяти, значение которой будет рассматриваться как последовательность бит, подлежащих сдвигу.

Счетчиком для процессора 8086 может быть либо 1, либо CL (для сдвига учитываются только младшие 5 бит). Начиная с процессора 186, вместо единицы можно указывать любую байтовую беззнаковую константу (учитываются только младшие 5 бит).

2.7.8.1. Команды логического сдвига SHL и SHR

Формат: **shl** ***Приемник, Счетчик***

shr ***Приемник, Счетчик***

Последовательность бит *Приемника* сдвигается влево (вправо) на число позиций, определяемое счетчиком, причем каждый бит, вытесняемый за пределы ячейки (байта или слова) последовательно записывается во флаг переноса CF, а недостающие биты заполняются нулями.

При сдвиге слов в памяти они сдвигаются так, как если бы хранились в памяти не "перевернутыми", т. е. команды сдвига автоматически учитывают "перевернутость" слов в памяти.

Сдвиг влево на *n* позиций эквивалентен умножению на 2^n для чисел без знака и со знаком (если, конечно, не происходит вытеснения значащих битов).

Сдвиг вправо на *n* позиций эквивалентен делению нацело на 2^n **только для чисел без знака**.

Умножения и деления при помощи сдвигов выполняются быстрее, чем при помощи команд MUL и DIV.

2.7.8.2. Команды арифметического сдвига SAL и SAR

Команда SAL является синонимом команды SHL.

Команда SAR действует почти так же, как команда SHR, но по окончании сдвига в самый левый (знаковый) бит заносится его первоначальное значение.

Сдвиг вправо на *n* позиций по команде SAR эквивалентен делению нацело на 2^n для чисел без знака и со знаком. Однако для отрицательных чисел результат этого деления будет отличаться от результата команды IDIV: команда IDIV округляет результат до ближайшего меньшего по абсолютной величине целого числа, а команда SAR округляет результат до ближайшего большего по абсолютной величине целого числа:

```
mov    AX,-3
mov    CL,2
idiv   CL              ; AL=-1
mov    AL,-3
sar    AL,1            ; AL=-2
```

2.7.8.3. Команды циклического сдвига ROL, ROR и RCL, RCR

Команды циклического сдвига ROL и ROR действуют так же, как команды SHL и SHR, но каждый вытесняемый бит попадает в ячейку "с другой стороны".

Команды циклического сдвига через флаг переноса RCL и RCR действуют так же, как команды ROL и ROR, но каждый вытесняемый бит попадает во флаг переноса CF, а бит из CF попадает в ячейку "с другой стороны".

2.7.9. Команды модификации флагов и команда холостого хода

У команд модификации флагов операндов нет. Эти команды не меняют значений других флагов.

STC (SeT Carry flag) — устанавливает CF в 1.
CLC (CLear Carry flag) — устанавливает CF в 0.
CMC (CoMplement Carry flag) — инвертирует CF.
STD (SeT Direction flag) — устанавливает DF в 1.
CLD (CLear Direction flag) — устанавливает DF в 0.
STI (SeT Interrupt Enable flag) — устанавливает IF в 1 (т. е. внешние прерывания разрешаются).
CLI (CLear Interrupt Enable flag) — устанавливает IF в 0 (т. е. внешние прерывания запрещаются).

По команде STI прерывания разрешаются не сразу после этой команды, а после следующей за ней.

По команде без операндов **NOP** (No OPeration) не производится никаких действий (за исключением увеличения значения регистра указателя команд IP).

2.7.10. Команды обработки цепочек

Будем называть *цепочкой байт или слов* некоторое количество байт или слов, непрерывно размещенных в памяти (при использовании расширенных регистров обрабатываются и цепочки двойных слов).

Цепочки можно обрабатывать в циклах при помощи рассмотренных выше команд. Однако имеются специальные команды, позволяющие обрабатывать цепочки более удобно и эффективно.

Все команды обработки цепочек имеют общие особенности:

- Мнемокоды команд имеют постфикс **B**, **W** или **D** для цепочек байтов, слов и двойных слов (далее цепочки двойных слов рассматривать не будем).
- С этими командами (и только с ними) могут применяться префиксы повторения **REP**—повторять, **REPE**—повторять пока равно, **REPNE**—повторять пока не равно, **REPZ**—повторять пока ноль, **PERNZ**—повторять пока не ноль.
- Эти команды не имеют явных операндов.
- Адрес элемента цепочки-источника задается парой **DS:SI** адрес элемента цепочки-приемника (если он есть) задается парой **ES:DI**.
- Действие любой команды без префикса повторения заключается в обработке элемента цепочки (или пары соответствующих элементов) с последующей перенастройкой на обработку следующего или предыдущего элемента.
- Направление обработки цепочек (т. е. увеличение или уменьшение значений *SI* и *DI*) выбирается в зависимости от значения флага **DF** (*0*-прямое, 1-- обратное направление).

- При применении префиксов повторения повторение осуществляется столько раз, сколько определено в регистре *CX* Кроме того, при префиксах *REPZ* и *REPNZ* повторение происходит в зависимости от значения флага *ZF*, а при префиксах *REPE, REPNE* — при равенстве или неравенстве соответствующих элементов.

- При обработке цепочек байт значения *SI* и *DI* изменяются (возрастают или убывают в зависимости от направления обработки) на 1, при обработке цепочек слов значения *SI* и *DI* изменяются на 2.

- Все рассмотренные команды действуют для процессора 8086. Имеются еще команды считывания цепочек из порта и записи в порт *INSB, INSW, OUTSB, OUTSB*, действующие для процессора 80186, но мы не будем рассматривать эти команды.

Следует отметить, что мы можем не применять префиксы повторения, а организовывать циклы с командами обработки цепочек «кустарно», но при этом эффективность сильно снижается, а длина кода возрастает.

2.7.10.1. Пересылка цепочек MOVSB, MOVSW

Производится пересылка байта или слова из источника по адресу *DS:SI* в приемник по адресу *ES:DI*, затем *SI* и *DI* увеличиваются (при *DF*=0) или уменьшаются (при *DF=1*) на 1 (для *MOVSB*) или на 2 (для *MOVSW*).

Эти команды не меняют флагов, поэтому нет смысла применять префиксы повторения *REPZ, REPNZ, REPE, REPNE* (точнее, они будут действовать точно так же, как префикс *REP*).

Пример (пересылка массива в графический видеобуфер):

```
        .data
a       db      64000 dup(1)            ;Цепочка-источник
        .code
Entr:   push    @data
        pop     DS                      ;Загрузка DS
        mov     AX,0A000h
        mov     ES,AX           ;Настройка ES на видеопамять
```

```
cld                ;DF=0, направление обработки прямое
mov   CX,64000                     ;Число повторений
xor   SI,SI
xor   DI,DI
rep   movsb                                  ;Пересылка
```

2.7.10.2. Сравнение цепочек CMPSB, CMPSW

Производится сравнение байт или слов по адресам *DS:SI* и *ES:DI*, и по результатам этого сравнения выставляются флаги (аналогично команде *CMP*), затем *SI* и *DI* увеличиваются (при *DF*=0) или уменьшаются (при *DF=1*) на 1 (для *CMPSB*) или на 2 (для *CMPSW*).

2.7.10.3. Сканирование цепочки SCASB, SCASW

Производится сравнение значения регистра *AL* (для *SCASB*) или регистра *AX* (для *SCASW*) с байтом или словом в памяти по адресу *ES:DI*, и по результатам этого сравнения выставляются флаги аналогично команде *CMP*, затем *DI* увеличивается (при *DF*=0) или уменьшаются (при *DF=1*) на 1 (для *SCASB*) или на 2 (для *SCASW*).

Команды применяются для поиска в цепочке первого вхождения элемента, равного заданному или отличному от заданного.

2.7.10.4. Запись значения регистра в элемент цепочки STOSB и STOSW

Производится запись в байт или слово в памяти по адресу *ES:DI* значения регистра *AL* (для *STOSB*) или *AX* (для *STOSW*), затем *DI* увеличивается (при *DF*=0) или уменьшаются (при *DF=1*) на 1 (для *STOSB*) или на 2 (для *STOSW*).

Эти команды не меняют флагов, поэтому нет смысла применять префиксы повторения *REPZ*, *REPNZ*, *REPE*, *REPNE* (точнее, они будут действовать точно так же, как префикс *REP*).

2.7.10.5. Загрузка элемента цепочки LODSB, LODSW в регистр

Производится загрузка регистра *AL* (для *LODSB*) или *AX* (для *LODSW*) байтом или словом из памяти по адресу *DS:SI*, затем *SI* увеличивается (при *DF*=0) или уменьшаются (при *DF=1*) на 1 (для *LODSB*) или на 2 (для *LODSW*).

Команды значений флагов не меняют.

Применять какие-либо префиксы повторения с этими командами бессмысленно.

Часто одна из этих команд применяется в паре с соответствующей командой *STOSB* или *STOSW* в «кустарно» организованном цикле по схеме: «загружаем очередной элемент цепочки в *AL/AX* — обрабатываем — записываем в цепочку».

2.7.11. Процедуры

При применении процедур возникают следующие вопросы:

Как вызывать процедуру? Ответ на этот вопрос приведен в разделе 2.7.5.4.

Как происходит возврат из процедуры? Ответ на этот вопрос также приведен в разделе 2.7.5.4.

Где располагать описание процедуры? Процедуру можно располагать, в принципе, где угодно, но так, чтобы на нее передавалось управление только по вызову и никак иначе. Наилучшее место расположения процедуры в простой программе — непосредственно перед первой выполняемой командой программы (в приведенных примерах — перед строкой с меткой Entry). Можно также располагать процедуры в отдельном программном сегменте команд (при моделях памяти MEDIUM, LARGE, HUGE).

Как описывается процедура?
Формат описания процедуры:

```
Имя    proc   Спецификация
              Тело процедуры
              ret
Имя    endp
```

В качестве Спецификации может быть указано **NEAR** или **FAR**. Если *Спецификация* не указана, по умолчанию полагается **NEAR**. При спецификации **NEAR** процедуру можно вызывать только из того программного сегмента, в котором она описана. При спецификации **FAR** процедуру можно вызывать из любого программного сегмента (вместо **ret** применяется **retf**).

Следует помнить, что в отличие от языков высокого уровня, метки в Ассемблере не локализованы в процедурах, и поэтому они должны быть уникальными.

Хорошей привычкой является сохранение всех РОН (при помощи команды PUSHA) и регистра флагов (при помощи команды PUSHF) в стеке перед вызовом процедуры с последующим восстановлением этих регистров (при помощи команд POPA и POPF) непосредственно перед выходом из процедуры. Конечно, это не всегда обязательно, но дает возможность возвратиться в вызываемую программу точно при таком же состоянии процессора, какое было до вызова и избежать нежелательных побочных эффектов.

Если процедура является одновременно функцией, т. е. возвращает нечто через некоторые регистры в вызывающую программу, то пару PUSHA–POPA использовать нельзя, и приходится сохранять и восстанавливать «испорченные» регистры индивидуально (при помощи команд PUSH--POP).

Если процедура не нуждается в параметрах при вызове, не должна возвращать значений в место вызова и не использует локальных переменных, то применение такой процедуры особых вопросов вызывать не должно.

С другой стороны, в Ассемблере нет никаких специальных средств для реализации перечисленных особенностей вызова процедур. Для их реализации (например, для передачи параметров и для возврата значений в место вызова) необходимо использовать все те же РОН и в особенности стек.

2. 7.11.1. Упрощенное описание процедур при помощи конвенций C, PASCAL и STDCALL

Пусть нам необходимо создать процедуру с тремя параметрами *x*, *y* и *z*, возвращающую результат в регистре AX (как это обыч-

но принято). Мы можем затолкнуть параметры в стек в их естественной последовательности и вызвать процедуру:

```
        push   x
        push   y
        push   z
        call   MyProc
```

При этом описание процедуры должно иметь вид наподобие следующего:

```
MyProc   proc
      push    BP     ;Сохраняем BP, который будем использовать
      mov     BP,SP    ;Настройка BP для доступа к параметрам
zl    equ     [BP+4]        ;В стеке значение BP и адрес возврата!
yl    equ     [BP+6]
xl    equ     [BP+8]
      ...                ;Текст процедуры с параметрами xl, yl и zl
      pop     BP                ;Восстанавливаем старое значение BP
      ret     6     ;Возврат с уничтожением трех элементов стека
MyProc   endp
```

Такой способ описания и вызова процедур, при котором ***параметры передаются через стек в прямой последовательности и стек очищается внутри процедуры***, характерен для языков высокого уровня PASCAL, MODULA2, BASIC, FORTRAN, ADA и некоторых других.

Для упрощенного описания и вызова такой процедуры можно записать вызов:

```
        call   MyProc(x,y,z)
```

и описание

```
MyProc   proc   PASCAL,x:word,y:word,z:word
          ...              ;Текст процедуры с параметрами x, y и z
          ret         ;Фактически будет выполнена команда ret 6
MyProc   endp
```

Все остальные команды, имеющиеся в первой программе, и отсутствующие во второй, будут добавлены автоматически.

Однако для выполнения той же задачи мы можем поступить по-другому, а именно, затолкнуть параметры в стек в обратной последовательности, вызвать процедуру и по возвращении из нее в вызывающей программе очистить стек:

```
        push   z
        push   y
        push   x
        call   MyProc
        add    SP, 6    ;Выталкиваем ненужные параметры из
стека
```

При этом описание процедуры должно иметь вид наподобие следующего:

```
MyProc    proc
    push   BP       ;Сохраняем BP, который будем использовать
    mov    BP,SP    ;Настройка BP для доступа к параметрам
x1  equ    [BP+4]      ;В стеке значение BP и адрес возврата!
y1  equ    [BP+6]
z1  equ    [BP+8]
    ...                ;Текст процедуры с параметрами x1, y1 и z1
    pop    BP          ;Восстанавливаем старое значение BP
    ret                                               ;Возврат
MyProc    endp
```

Такой способ описания и вызова процедур, при котором ***параметры передаются через стек в обратной последовательности и стек очищается в вызывающей программе***, характерен для языков высокого уровня C, C++, JAVA, PROLOG и некоторых других.

Для упрощенного описания и вызова такой процедуры можно записать вызов:

```
        call   MyProc(x,y,z)
```

и описание

```
MyProc   proc   C,x:word,y:word,z:word
    ...                  ;Текст процедуры с параметрами x, y и z
    ret
MyProc   endp
```

Все остальные команды, имеющиеся в первой программе и отсутствующие во второй, будут добавлены автоматически.

Наконец, для выполнения той же задачи мы можем поступить третьим способом, а именно, затолкнуть параметры в стек в обратной последовательности, как в C, но очищать стек не в вызывающей программе, а в самой процедуре, как в PASCAL:

```
        push   z
        push   y
```

```
        push   x
        call   MyProc
```

При этом описание процедуры должно иметь вид наподобие следующего:

```
MyProc  proc
     push   BP     ;Сохраняем BP, который будем использовать
     mov    BP,SP    ;Настройка BP для доступа к параметрам
x1   equ    [BP+4]         ;В стеке старое BP и адрес возврата!
y1   equ    [BP+6]
z1   equ    [BP+8]
     ...                ;Текст процедуры с параметрами x1, y1 и z1
     pop    BP              ;Восстанавливаем старое значение BP
     ret   6     ;Возврат с уничтожением трех элементов стека
MyProc  endp
```

Такой способ описания и вызова процедур, ***при котором параметры передаются через стек в обратной последовательности, а стек очищается внутри процедуры***, характерен для программирования под Windows (win32 API).

Для упрощенного описания и вызова такой процедуры можно записать вызов:

```
        call   MyProc(x,y,z)
```

и описание

```
MyProc  proc   STDCALL,x:word,y:word,z:word
     ...                ;Текст процедуры с параметрами x1, y1 и z1
     ret                ;Фактически будет выполнена команда ret 6
MyProc  endp
```

Все остальные команды, имеющиеся в первой программе и отсутствующие во второй, будут добавлены автоматически. В описании процедуры параметр STDCALL можно опустить, поскольку он предполагается по умолчанию.

Существуют и другие конвенции. Например, в Ассемблере Watcom предусмотрена конвенция, в соответствии с которой первые 4 параметра передаются через регистры AX, DX, BX, CX (именно в таком порядке), а 5-й и следующие параметры передаются через стек согласно обычной С-конвенции.

2.7.12. Команды ввода/вывода. Прерывания

2.7.12.1. Команды ввода и вывода IN и OUT

Формат: **in** *Приемник, Источник*

out *Приемник, Источник*

Приемником команды IN и источником команды OUT могут быть только AL, AX, EAX.

Источники команды IN и команды OUT — номер порта (не более 255), задаваемый константой или значением регистра DX.

Однако осуществление ввода/вывода непосредственно при помощи этих команд — дело достаточно сложное. Одному внешнему устройству и его контроллеру соответствует не один, а несколько портов отдельно для управления устройством и для передачи данных. Для организации ввода/вывода необходимо знать достаточно сложный сценарий обмена данными, технические характеристики конкретного устройства, протоколы связи с ним, порядок опроса портов и т. п.

Поэтому обычно операции обмена с внешними устройствами широкого применения оформляются в виде процедур операционной системы, вызов которых осуществляется через прерывания.

2.7.12.2. Концепция прерывания

Прерыванием называется реакция процессора на некоторое событие, заключающаяся в том, что процессор приостанавливает выполнение текущей программы, записывает в ее стек значения регистров FLAGS, CS, IP, обнуляет флаги IF и TF и переключается на выполнение другой программы, называемой **процедурой обработки прерывания**. По окончании этой процедуры процессор возвращается к выполнению прерванной программы. Особая трудность здесь заключается в том, что прерывания поступают независимо от выполняемой программы, в любой произвольный момент. Кроме того, запросов на прерывания может быть несколько, и из них необходимо формировать очередь.

Прерывания могут быть **маскируемыми** и **немаскируемыми**. Маскируемые прерывания блокируются занесением 0 в флаг IF. Немаскируемые (самые важные) прерывания заблокировать невоз-

можно. Для маскируемых и немаскируемых прерываний у процессора есть два отдельных физических входа (INTR и NMI).

Прерывания можно разделить на **внешние** (поступающие от внешних устройств) и **внутренние**. Внутренние в свою очередь делятся на **программные**, т. е. инициируемые самой программой пользователя для обращения к функциям DOS и BIOS, и так называемые **исключения** (exceptions) — особые ситуации, возникающие в процессоре (например, деление на ноль).

На уровне электроники внешние прерывания организованы следующим образом. Имеется контроллер прерываний (микросхема i8259A), выполняющий несколько важных функций: получение сигналов на прерывания от внешних устройств, маскировка (т. е. запрещение обработки) некоторых прерываний, арбитраж приоритетов прерываний и формирование очереди запросов на прерывание. Контроллер имеет три 8-разрядных регистра и 8 входов от внешних устройств, называемых IRQ0, IRQ1, ...IRQ7 (Interrupt ReQuest), выход INT на процессор (соединяемый со входом процессора INTR) и вход обратной связи от процессора INTA, по которому процессор подтверждает начало обработки прерывания.

Запрос на прерывание, поступающий на некоторый вход IRQ, устанавливает в 1 соответствующий бит 8-разрядного регистра фиксации запросов прерываний (IRR). Если соответствующее прерывание не замаскировано 8-разрядным регистром маски (IMR может программироваться) и если процессор не занят обработкой прерывания высшего или равного приоритета (что определяется 8-разрядным регистром ISR), запрос на прерывание поступает с выхода INT на вход прерываний процессора INTR, и процессор подтверждает по линии INTA.

Один контроллер прерываний имеет 8 входов (IRQ), чего явно недостаточно для обслуживания внешних устройств современного компьютера (таймер, часы реального времени. клавиатура, гибкий и жесткий диски, мышь, последовательные и параллельные порты и проч.).

Однако несколько контроллеров прерываний могут быть включены последовательно. На современных компьютерах (начиная с 286) один (ведущий) контроллер прерываний подключен непосредственно к процессору, а второй (ведомый) своим выходом INT подключен ко входу IRQ2 ведущего контроллера. Итого

получается 15 входов прерываний от IRQ0 до IRQ15 (IRQ2 не может быть использован).

IRQ	Устройство	Приоритет
0	Таймер	2
1	Клавиатура	3
2	Выход INT ведомого контроллера	
3	COM2	12
4	COM1	13
5	LPT2	14
6	Контроллер гибкого диска	15
7	LPT1	16
8	Часы реального времени (CMOS)	4
9	Свободен	5
10	Свободен	6
11	Свободен	7
12	Свободен	8
13	Свободен	9
14	Контроллер жесткого диска	10
15	Свободен	11

При подаче сигнала на прерывание от контроллера к процессору по линии INTR INT, одновременно по линиям данных из контроллера прерываний в процессор поступает номер **вектора прерывания**, который образуется путем сложения IRQ с некоторым базовым номером, который присваивается BIOSом контроллеру в процессе загрузки (значения по умолчанию 08*h* для ведущего контроллера и 70*h* для ведомого). Таким образом, например, номер вектора прерывания для клавиатуры будет 08+1=9*h*, для жесткого диска 70*h*+6=76*h*.

Контроллер прерываний допускает перепрограммирование для установки различных режимов формирования очереди запросов, изменения приоритетов прерываний, изменения базовых номеров контроллеров. Такое перепрограммирование осуществляется через два байтовых порта 20*h* и 21*h*. По умолчанию IRQ отдельного контроллера имеют приоритеты в соответствии с их номерами (IRQ0 — наивысший, IRQ7 — наинизший).

Итак, сформирован и поступил в процессор номер вектора прерывания. Как уже говорилось, начальный участок памяти отводится на 256 4-байтовых векторов прерываний. В векторах прерываний хранятся смещения и сегменты ("перевернуто"!) процедур обработки соответствующих прерываний. Процессор записывает в стек выполнявшейся программы значения регистров FLAGS, CS, IP, обнуляет флаги прерываний и трассировки IF и TF (чтобы заблокировать обработку других прерываний и трассировку внутри процедуры обработки прерывания) и передает управление в процедуру обработки прерывания, адрес которой определен вектором прерывания (если, конечно, прерывание не заблокировано флагом IF=0).

Внутренние прерывания (исключения), инициируемые процессором, также имеют свои номера векторов прерывания, например, деление на ноль имеет вектор прерывания номер 0, попытка выполнения несуществующей команды имеет номер 6. Вызов процедуры обработки прерывания для них проходит точно так же.

В конце каждой процедуры прерывания выполняется команда **IRET**. Она действует так же, как команда RET, а кроме того, автоматически восстанавливает из стека старое значение регистра флагов.

Особо важную роль для программирования выполняют прерывания, инициированные программой пользователя по команде

int ***Номер_прерывания***

Такие прерывания применяются в первую очередь для вызова системных функций DOS и BIOS. На самом деле прерывания с номерами векторов 21*h* (функции DOS) и 10*h* (функции BIOS) являются целыми библиотеками со множеством функций (номера которых загружаются в AH перед вызовом прерывания).

При помощи команды INT можно имитировать также внешнее прерывание или исключение. Необходимо особо подчеркнуть, что обработка прерываний осуществляется по единому сценарию независимо от причины, его вызвавшей.

По существу вызов прерывания по команде INT отличается от вызова far-процедуры по команде CALL только тем, что некоторые действия (сохранение и последующее восстановление FLAGS, обнуление IF и TF) производятся автоматически. Если внутри

процедуры обработки прерывания какие-либо РОН "портятся", они сначала сохраняются в стеке, а потом восстанавливаются.

Именно из-за активного использования стека в процессе обработки прерываний сегмент стека должен быть определен даже в программе, не использующей стек явно. Если сегмент стека не определен, стек все равно организуется "на дне" сегмента команд, и по мере роста он может начать "затирать" команды.

2.7.12.3. Функции DOS-прерывания 21*h* и BIOS-прерывания 10*h*

2.7.12.3.1. Функции DOS-прерывания 21h

Код функции заносится в регистр AH.

00	Завершение программы (аналогично INT 20*h*).
01	Ввод символа с клавиатуры с эхом (ASCII-код символа в AL).
02	Вывод символа на экран.
03	Ввод символа из асинхронного коммуникационного канала.
04	Вывод символа на асинхронный коммуникационный канал.
05	Вывод символа на печать.
06	Прямой ввод с клавиатуры и вывод на экран.
07	Ввод с клавиатуры без эха и без проверки Ctrl/Break.
08	Ввод с клавиатуры без эха с проверкой Ctrl/Break (ASCII-код символа в AL).
09	Вывод строки символов на экран (адрес стороки в DS:DX, в конце строки символ $).
0A	Ввод с клавиатуры с буферизацией (адрес буфера в DS:DX).
0B	Проверка наличия ввода с клавиатуры (AL=0 -- буфер пуст, AL=FF*h* -- буфер не пуст).
0C	Очистка буфера ввода с клавиатуры и запрос на ввод.
0D	Сброс диска.
0E	Установка текущего дисковода.
0F	Открытие файла через FCB.
10	Закрытие файла через FCB.

11 Начальный поиск файла по шаблону.
12 Поиск следующего файла по шаблону.
13 Удаление файла с диска.
14 Последовательное чтение файла.
15 Последовательная запись файла.
16 Создание файла.
17 Переименование файла.
18 Внутренняя операция DOS.
19 Определение текущего дисковода.
1A Установка области передачи данных (DTA).
1B Получение таблицы FAT для текущего дисковода.
1C Получение FAT для любого дисковода.
21 Чтение с диска с прямым доступом.
22 Запись на диск с прямым доступом.
23 Определение размера файла.
24 Установка номера записи для прямого доступа.
25 Установка вектора прерывания.
26 Создание программного сегмента.
27 Чтение блока записей с прямым доступом.
28 Запись блока с прямым доступом.
29 Преобразование имени файла во внутренние параметры.
2A Получение даты (CX — год, DH — месяц, DL — день).
2B Установка даты.
2C Получение времени (CH — часы, CL — минуты, DH — секунды,DL — 1/100 секунд).
2D Установка времени.
2E Установка/отмена верификации записи на диск.
2F Получение адреса DTA в регистровой паре ES:BX.
30 Получение номера версии DOS в регистре AX.
31 Завершение программы, после которого она остается резидентной в памяти.
33 Проверка Ctrl/Break.
35 Получение вектора прерывания (адреса подпрограммы).
36 Получение размера свободного пространства на диске.
38 Получение форматов различных государств.

39 Создание подкаталога (команда MKDIR).
3A Удаление подкаталога (команда RMDIR).
3B Установка текущего каталога (команда CHDIR).
3C Создание файла без использования FCB.
3D Открытие файла без использования FCB.
3E Закрытие файла без использования FCB.
3F Чтение из файла или ввод с устройства.
40 Запись в файл или вывод на устройство.
41 Удаление файла из каталога.
42 Установка позиции для последовательного доступа.
43 Изменение атрибутов файла.
44 Управление вводом/выводом для различных устройств.
45 Дублирование файлового номера.
46 "Склеивание" дублированных файловых номеров.
47 Получение текущего каталога.
48 Выделение памяти из свободного пространства.
49 Освобождение выделенной памяти.
4A Изменение длины блока выделенной памяти.
4B Загрузка/выполнение программы (подпроцесса).
4C Завершение подпроцесса с возвратом управления.
4D Получение кода завершения подпроцесса.
4E Начальный поиск файла по шаблону.
4F Поиск следующего файла по шаблону.
54 Получение состояния верификации.
56 Переименование файла.
57 Получение/установка даты и времени изменения файла.
59 Получение расширенного кода ошибки.
5A Создание временного файла.
5B Создание нового файла.
5C Блокирование/разблокирование доступа к файлу.
62 Получение адреса префикса программного сегмента (PSP).

2.7.12.3.2. Экранные функции BIOS-прерывания 10h

Код функции заносится в регистр AH.
00 Установка текстовых и графических режимов.

01 Установка типа и размера курсора.

02 Позиционирование курсора.

03 Считывание позиции курсора.

04 Считывание позиции светового пера.

05 Выбор активной видеостраницы.

06 Прокрутка активной страницы вверх (AL=количество строк прокрутки).

07 Прокрутка активной страницы вниз (AL=количество строк прокрутки).

08 Считывание атрибута/символа в текущей позиции курсора (AL=символ, AH=атрибут).

09 Вывод атрибута/символа в текущую позицию курсора (AL=ASCII–код символа, AH=атрибут).

0A Запись символа в текущую позицию курсора (AL=ASCII-код символа).

0B Установка палитры.

0C Запись пикселя.

0D Считывание пикселя.

0E Вывод в режиме телетайпа.

0F Получение текущего видеорежима.

10 Установка палитры (специальные режимы).

13 Вывод символьной строки.

2.7.13. Макросы и макроопределения

Макросредствами какого-либо языка программирования являются средства обработки программы на этом языке как текста еще до поступления программы на компиляцию. Можно представлять себе некоторый "макроимитатор", на вход которого поступает *текст программы*, включающий в себя некоторые макроопределения, а на выходе получается *также текст программы*, уже обработанный в соответствии с этими макроопределениями. Конечно, не всегда этот "макроимитатор" существует физически отдельно от компилятора, чаще всего они объединены, однако можно представлять себе, что компилятор работает в два прохода: в первый проход "макроимитатор" обрабатывает текст в соответствии с макроопределениями, а во второй

проход происходит уже "настоящая" компиляция — перевод текста программы в машинный код.

Примерами макросредств в языках высокого уровня являются директивы #include и #define.

Директивы Ассемблера EQU и INCLUDE можно также в некоторой степени отнести к макросредствам Ассемблера.

В Ассемблере предусмотрено значительное количество макросредств. такие, как средства условной компиляции, различные блоки повторения и проч., но мы их рассматривать не будем.

Рассмотрим только наиболее мощное макросредство Ассемблера, называемое *макросом* или *макрокомандой.* Макросы в большой степени упрощают программирование на Ассемблере и делают программу гораздо более наглядной.

Макроопределением назовем совокупность нескольких команд и/или директив Ассемблера, объединенных под одним именем.

Формат макроопределения:

Имя **macro** *Форм_Парам1, Форм_Парам2, Форм_Парам3 ...*

. . .
ТелоМакроопределения
. . .
endm

В тело макроопределения могут входить любые директивы и команды Ассемблера (за исключением самих макросредств). Необязательные формальные параметры макроопределения (в любом числе) локализованы в теле макроопределения (те же имена можно применять в других местах программы, конечно, в другом смысле).

Макроопределения внешне похожи на описания процедур и функций в языках высокого уровня (функция в **C++**, описанная как **inline**, является почти полным аналогом макроопределения в Ассемблере). Однако по существу макроопределения существенно отличаются как от процедур и функций в языках высокого уровня (за исключением inline-функций). так и от процедур в Ассемблере. Процедуры и функции *вызываются* из других мест программы (т. е. организуются соответствующие передачи управления), макроопределения *вписываются* в места вызова

просто как куски текста (правда, с заменой формальных параметров на фактические).

После того, как макроопределение описано (выше по программе), его можно "вызывать" как некоторую новую команду Ассемблера (**макрос**):

Имя_Макро *Факт_Парам1, Факт_Парам2, Факт_Парам3 ...*

Число фактических параметров макроса не обязательно должно быть равно числу формальных параметров макроопределения. Если фактических параметров меньше, считается, что не указанные фактические параметры имеют значение "пробел", если фактических параметров больше, лишние игнорируются.

Макроопределения лучше всего ставить впереди всей программы (или записывать в отдельный файл, включаемый по директиве INCLUDE).

Пример:

```
Push_4      macro a,b,c,d
            push  a
            push  b
            push  c
            push  d
            endm
```

Вызов в программе:

```
Push_4   AX,[BX],ES:[13h],memcell
```

Нельзя забывать о том, что макроопределение рассчитано на многократное копирование в различные места программы. Следовательно, если мысленно представить, что все макросы в программе заменены на соответствующие им макроопределения, результирующая программа должна быть *правильной* программой на Ассемблере.

Отсюда, в частности, следует специфический подход к меткам в макроопределениях. Если не принимать никаких мер, то при многократном копировании макроопределения в результирующей программе окажется несколько одинаковых меток. Чтобы этого не случилось, необходимо описать предполагаемую к использованию метку как локальную: **local** ***Метка***

Тогда при многократном копировании макроопределения в каждом из фрагментов будут уникальные метки ??xxxx (где xxxx — порядковый номер локальной метки в программе).

Пример (задержка на 5×*m* тактов для процессора *386*):

```
MyDelay     macro m
            local MyLabel
            mov   CX, m
MyLabel:    loop  MyLabel
            endm
```

Если теперь в программе написать эту макрокоманду два раза:

```
        MyDelay     100     ; Задержка на 500 тактов
        ...
        MyDelay     200     ; Задержка на 1000 тактов
```

фактически будет сгенерирован следующий фрагмент программы:

```
                    mov   CX, 100
        ??0000:     loop  ??0000
                    ...
                    mov   CX, 200
        ??0001:     loop  ??0001
```

Обратим внимание на некоторые тонкости использования фактических параметров в макроопределениях. Во-первых, фактические параметры в макрокоманде могут перечисляться как через запятую, так и через пробел.

Итак, при перечислении фактических параметров запятая, пробел и точка с запятой (начало комментария) имеют специальный смысл разделителей фактических параметров. Однако может случиться, что внутри самого фактического параметра могут быть запятые и/или пробелы, например, мы хотим в качестве фактического параметра передать макросу слово New York. Компилятор поймет это, как два стоящих рядом фактических параметра. Чтобы этого не было, необходимо заключить фактический параметр в угловые скобки: <New York>.

Кроме того, знак ! отменяет специальное значение символа, стоящего за ним, например, запятая в паре символов !, трактуется уже не как разделитель, а просто как символ запятой. В частности, чтобы использовать внутри фактического параметра

символ восклицательного знака, его необходимо удвоить (т. е. отменить его специальное значение).

Иногда неоднозначности могут возникнуть в макроопределениях с применением формальных параметров. Например, в приведенном выше примере с макросом MyDelay буква **m** в мнемокоде **mov**, конечно, не заменяется на значение фактического параметра. Однако бывают случаи, когда такая замена желательна. В этом случае в теле макроопределения следовало бы написать **&m&ov** (конечно, в данном случае получилась бы бессмыслица).

Итак, если формальный параметр текстуально встречается внутри литерала, он формальным параметром не считается. При необходимости отмены этого правила формальный параметр внутри литерала необходимо заключить в знаки **&**.

Вообще говоря, в макроопределениях можно использовать другие макросы и даже вызывать макросы рекурсивно, однако вряд ли можно это рекомендовать.

В заключение сравним процедуры и макросы. Хотя внешне они похожи, механизм их реализации совершенно различен. Макроопределения копируются в места макросов еще до компиляции программы. Процедуры компилируются в отдельные участки программы, генерируются передачи управления на эти участки программы и возвраты управления в точки вызова.

Отсюда можно сделать вывод, что макросы обеспечивают несколько более эффективный машинный код (за счет отсутствия команд CALL и RET, отсутствия необходимости запоминания и восстановления значений регистров и проч.), чем соответствующие процедуры. Макросы также несколько более удобны в применении за счет наличия формальных параметров и локализованных меток.

С другой стороны, при использовании макросов код программы может многократно удлиняться (за счет многократного копирования макроопределений).

Поэтому можно дать следующие рекомендации:

- если критичным является быстродействие (эффективность) программы, следует применять макросы,

- если критичным является размер программы, следует применять процедуры,
- более короткие и простые отрезки программы следует оформлять как макросы, более длинные и сложные — как процедуры.

3. Приемы программирования на Ассемблере

3.1. Массивы и структуры

В языках высокого уровня *массивом называется переменная, представляющая собой совокупность нескольких элементов **одного типа***. Эта совокупность имеет одно имя, а доступ к ее отдельным элементам осуществляется при помощи индекса.

В Ассемблере массивом можно назвать несколько подряд идущих в памяти байт, слов или двойных слов, но все элементы массива должны быть либо байтами, либо словами, либо двойными словами, т. е. иметь ***одинаковую длину***. В качестве имени массива используется символическое имя адреса (смещения) первого байта первого элемента массива. Массивы байт иногда называют строками.

Следует особо подчеркнуть: о том, что данная последовательность элементов является массивом, знает только программист, компьютер об этом "не знает". Поэтому, естественно, при компиляции и выполнении программы не производится какого-либо контроля на выход за пределы массива и проч. Например, программист может определить массив из 10 элементов, а затем обратиться к 15-му элементу этого массива — ответственность за такое обращение будет целиком лежать на программисте.

В целом можно констатировать, что подход к массивам в Ассемблере очень близок к подходу к массивам в языке С: весь массив задается адресом первого элемента и размером элементов, "конец" массива никак не фиксируется.

В языках высокого уровня *структурой (или записью) называется переменная, представляющая собой совокупность нескольких элементов (в общем случае) **разных типов***. Эта совокупность имеет одно имя, а доступ к отдельным элементам (называемым *полями*) осуществляется при помощи составного имени (составленного из имени самой структуры и имени поля).

Аналогично в Ассемблере *структурой можно назвать несколько подряд идущих в памяти байт, слов или двойных слов, причем элементы (поля) могут быть и байтами, и словами, и двойными словами, т. е. иметь **различную длину***. В качестве имени струк

туры используется символическое имя адреса (смещения) первого байта структуры, а в качестве имен полей используются символические имена смещений полей относительно имени структуры.

Следует подчеркнуть разницу между описаниями массивов и структур в Ассемблере (которая имеется, впрочем, и в языках высокого уровня). Когда мы описываем массив, компилятор выделяет память для него. Со структурами дело обстоит сложнее. Сначала мы должны описать тип структуры, т. е. указать перечень полей, из которых она состоит. На этом этапе память компилятором еще не выделяется. После того, как тип структуры описан, мы можем описывать переменные, являющиеся конкретными воплощениями (объектами, ипостасями, экземплярами) описанного выше типа структуры. Именно на этом этапе компилятор выделяет память под экземпляры структур.

3.1.1 Работа с массивами

Продемонстрируем работу с массивами на простейшем примере. Имеются два массива *x* и *y* с 10 элементами. Необходимо получить массив z, элементы которого являются суммами соответствующих элементов массивов *x* и *y*, а также скалярное произведение *x* и *y*.

```
      .model  small
      .stack  128
      .data
n     equ   10
x   dw   1,2,3,4,n-4 dup(1)      ; Массив:1,2,3,4,1,1,1,1,1,1
y   dw   5,6,7,8,9,n-5 dup(2)    ; Массив:5,6,7,8,9,2,2,2,2,2
z   dw   n dup (?)                          ;Для массива-суммы
s   dw   0                           ;Для скалярного произведения
      .code
Entr:  mov  AX,@data
      mov   DS,AX
      xor   SI,SI                       ;Обнуляем индекс-регистр
      mov   CX,n                        ;Загружаем счетчик цикла
Cycl:  mov  AX,x[SI]                                  ;AX:=xi
      add   AX,y[SI]                               ;AX:= xi +yi
      mov   z[SI],AX                                ;zi =xi +yi
```

```
    mov   AX,x[SI]                              ;AX:=xi
    imul  y[SI]                                 ;AX:= xi* yi
    add   s,AX                                  ;s:=s+ xi* yi
    add   SI,2                    ;Наращиваем индекс на 2
    loop  Cycl                           ;Повторить n раз
    mov   AH,4Ch
    int   21h
    end   Entr
```

Продемонстрируем работу с двумерными массивами. Пусть имеется матрица слов *a* размером 5×6. Необходимо ввести ее элементы. Если элементы матрицы обозначаются индексами i=0,1,2,3,4 и j=0,1,2,3,4,5, то, очевидно, смещение (i,j)-го элемента матрицы относительно ее начала будет 20*(2i)+(2j). (Двойки появились потому, что элементы матрицы — слова).

```
    include io.asm           ;Учебное расш. ввода/вывода
    .model  small
    .stack  128
    .data
m   equ   5
n   equ   6
a   dw    m dup( n dup(?))               ;Двумерный массив
Invit  db  "Next number: $"             ;Текст приглашения
    .code
Entr:  mov AX,@data
    mov   DS,AX
    lea   DX,Invit               ;Адрес строки приглаш. в DX
    xor   BX,BX                  ;Обнуляем регистр-модиф.
    mov   CX,m                   ;Загр. счетчик внутр. цикла
C_Ext:  push  CX                 ;Сохр. счетчик внешн. цикла
    mov   CX,n                   ;Загр. счетчик внутр. цикла
    xor   SI,SI                  ;Обнуляем индекс-регистр
C_Int:  outstr                   ;Вывод приглашения
    inint a[BX][SI]              ;Ввод очередного числа
    add   SI,2                   ;Наращиваем внутр. индекс
    loop  C_Int                  ;Повторить n раз
    add   BX,2*n                 ;Наращиваем внешн. индекс
    pop   CX                     ;Восст. счетчик внешн. цикла
    loop  C_Ext                  ;Повторить m раз
    mov   AH,4Ch
     int   21h
```

```
        end    Entr
```

Следует подчеркнуть, что для двойного индексирования мы можем применять пары регистров (BX,SI), (BX,DI), (BP,SI), (BP,DI), но не можем применять пары (SI,DI) и (BX,BP).

Оформим эту программу в виде процедуры. Будем предполагать, что значение *m* передается в регистре AX, значение *n* передается в регистре DX, адрес матрицы *a* передается в регистре BX. Заметим, что в этом случае нам не удобно размещать строку приглашения в сегменте данных, удобнее разместить ее вывод внутри процедуры.

```
InMatr  proc
        pusha                        ;Сохраняем значения РОН
        mov     CX,AX                ;Загр. счетчик внешн. цикла
L1:     push    CX                   ;Сохр. счетчик внешн. цикла
        mov     CX,DX                ;Загр. счетчик внутр. цикла
        xor     SI,SI                ;Обнуляем индекс-регистр
L2:     outch   '>'                  ;Вывод приглашения >
        inint   [BX][SI]             ;Ввод очередного числа
        add     SI,2                 ;Наращиваем внутр. индекс
        loop    L2                   ;Повторить n раз
        shl     DX,1                 ;DX:=2*n
        add     BX,DX                ;Наращиваем внешн. индекс
        shr     DX,1                 ;DX:=n
        pop     CX                   ;Восст. счетчик внешн. цикла
        loop    L1                   ;Повторить m раз
        popa                         ;Восстанавливаем РОН
        ret                          ;Возврат из процедуры
InMatr  endp
```

Теперь нетрудно написать процедуру вывода матрицы на экран при тех же самых предположениях.

```
OutMatr  proc
        pusha                        ;Сохраняем значения РОН
        mov     CX,AX                ;Загр. счетчик внешн. цикла
L3:     push    CX                   ;Сохр. счетчик внешн. цикла
        mov     CX,DX                ;Загр. счетчик внутр. цикла
        xor     SI,SI                ;Обнуляем индекс-регистр
```

```
L4:  outint [BX][SI],4          ;Вывод очередного числа
     add    SI,2                ;Наращиваем внутр. Индекс
loop L4     ;Повторить n раз
     newline                    ;Переход на новую строку
     shl    DX,1                ;DX:=2*n
     add    BX,DX               ;Наращиваем внешн. индекс
     shr    DX,1                ;DX:=n
     pop    CX                  ;Восст. счетчик внешнего цикла
     loop   L3                  ;Повторить m раз
     popa                       ;Восст. значения РОН
     ret                        ;Возврат из процедуры
OutMatr   endp
```

Для этих процедур данные могли бы быть представлены так:

```
m    equ    5
n    equ    6
a    dw     m dup (n dup(?))
```

Вызов этих процедур мог бы выглядеть так:

```
     mov    AX,m
     mov    DX,n
     lea    BX,a
     InMatr
     OutMatr
```

Напомним, что в Ассемблере предусмотрена целая группа команд для работы с цепочками, т. е. с одномерными массивами(команды *MOVS, CMPS, SCAS, STOS, LODS)*, позволяющие обрабатывать массивы гораздо более эффективно.

При работе со строками всегда возникает вопрос о создании "динамических" строк, т. е. строк переменной длины. Здесь Ассемблер не оказывает программисту никакой помощи, и программист должен сам позаботиться о создании таких строк.

Во всех случаях необходимо задать максимальную длину строки, т. е. описать массив байт, например:

```
MyString db 256 dup(?)     ;Строка не более 256 символов
```

Затем можно пойти, например, по пути языка PASCAL и хранить в первой ячейке этого массива его "динамическую" длину, а символы самой строки хранить в остальных байтах, причем следует считать, что все, что отстоит от начала массива на рас

стояние, большее "динамической" длины, к строке не относится. При изменении длины строки программист должен сам заботиться об изменении значения в ее первом байте.

Можно пойти по пути языка С и выделить некоторый символ "конца строки", который заведомо не будет встречаться внутри строки. При этом следует полагать, что все, что идет за первым символом "конца строки", к строке не относится. При изменении длины строки программист должен сам заботиться о перестановке символа "конца строки" на нужное место.

Оба эти способа организации "динамических" строк имеют свои очевидные преимущества и недостатки.

3.1.2. Структуры

Тип структуры имеет следующий **формат описания:**

```
Имя_типа        struc
                Описание_поля_1
                Описание_поля_2
                . . .
Имя_типа        ends
```

Описаниями полей являются знакомые нам директивы DB, DW и DD. Например:

```
Client  struc
Name            db      "* * * * * *"
Age             db      ?
Phone           db      8 dup(?)
Account         dd      ?
Year            dw      2000
Client          ends
```

В отличие от языков высокого уровня, имена полей не локализованы в структурах и должны быть уникальными. Поля внутри структур не могут быть структурами, т. е. вложенные структуры не допускаются.

Подчеркнем, что указанные здесь директивы DB, DW и DD являются только похожими на соответствующие директивы описаний, поскольку под них не выделяется память (они играют для компилятора только информационную роль). Значение 2000 в директиве этого примера DW является не значением для инициализации пе-

ременной YEAR, а *значением этого поля по умолчанию*. Таким образом можно сказать, что в данном примере поля AGE, PHONE и ACCOUNT не имеют значений по умолчанию, поле NAME имеет значение по умолчанию, равное "* * * * * *", поле YEAR имеет значение по умолчанию, равное 2000.

После того (ниже по программе), как тип структуры описан, можно описывать отдельные экземпляры такой структуры, например:

```
Smith  Client     <"John", ,56, 33500, 1998>
Lopez  Client     <"Diego", ,73, 523000, >
Strip  Client     <"Maryl", ,32, , >
```

Именно в этот момент происходит выделение памяти для полей, а следовательно, и для всего экземпляра структуры.

Значения для инициализации полей указываются в угловых скобках, причем число занятых в угловых скобках должно точно соответствовать числу полей. Если вместо некоторого значения для инициализации стоит пробел, соответствующее поле либо не инициализируется (если нет значения по умолчанию), либо инициализируется значением по умолчанию (если оно есть).

Нескалярные поля, т. е. поля, в описании которых использованы несколько операндов или конструкция повторения DUP (наподобие поля PHONE в примере), инициализации не подлежат.

Например, первый экземпляр структуры будет инициализирован значениями, указанными в угловых скобках, второй экземпляр структуры будет инициализирован значениями **"Diego", ?, 73, 523000, 2000**, третий экземпляр структуры будет инициализирован значениями **"Maryl", ?, 32, ?, 2000**.

Необходимо отметить, что имена полей являются фактически символическими именами смещений полей относительно начала структуры.

Напомним, что в Ассемблере имеется оператор TYPE, возвращающий длину обьекта в байтах. Этот оператор особенно ценен для структур, поскольку подсчет их длины вручную может быть затруднительным. Например, по команде

```
        mov    AX,type Smith
```

в регистр AX будет записано число 21.

Доступ к полям экземпляра структуры осуществляется, как и в языках высокого уровня, при помощи составных имен, которые

образуются из имени экземпляра структуры, точки и имени поля экземпляра структуры Например:

```
        mov    AL,Smith.Age
```

Очевидно, что составное имя есть символическое имя смещения данного поля данного экземпляра структуры относительно начала соответствующего сегмента. Это смещение образуется как сумма смещения начала экземпляра структуры и смещения данного поля относительно начала экземпляра структуры.

3.2. Реализация конструкций языков высокого уровня

3.2.1 Логические выражения

Алгебру булевских выражений Паскаля легко воспроизвести, введя булевские константы следующим образом:

```
        true  equ    11111111b             ; =0FFh
        false equ    00000000b             ; =0
```

Нетрудно проверить, что при этом команды NOT, AND, OR, XOR приводят к правильным результатам согласно традиционной алгебре логики. (Кстати, очевидно, что взять просто 1 в качестве **true** нельзя, поскольку, например, отрицание 1 есть 0FE*h*.)

В языке С любое арифметическое выражение считается истинным, если его значение отлично от нуля, и ложным в противном случае. Легко видеть, что команды NOT, AND, OR, XOR не приводят к правильным результатам.

Напишем макрос, например, для С-интерпретации логической операции AND (для байт). Макрос C_and с двумя операндами должен записывать в первый операнд значение 0FF*h*, если оба операнда не равны нулю, и должен записывать в первый операнд значение 0 в противном случае.

```
C_and macro a,b
      local L1,L2
      test  a,0FFh            ;Есть ли ненулевые биты в a?
      jz    L1
      test  b,0FFh            ;Есть ли ненулевые биты в b?
      jz    L1
      mov   a,0FFh
      jmp   L2
```

```
L1:    mov   a,0
L2:    nop
```

Аналогично можно построить остальные макросы для С-интерпретации логических операций NOT и OR.

3.2.2 Конструкция IF–THEN–ELSE и переключатель

Схема реализации условного оператора С (или аналогичного оператора Паскаля)

if (*Условие*) {*Операторы1*} else {*Операторы2*}

такова:

(Команды проверки условия)
jxxx ELSE
(Команды, соответствующие *Операторам1*)
jmp ENDIF
ELSE: (Команды, соответствующие *Операторам2*)
ENDIF: nop

Напомним еще раз, что метки во всей программе (точнее, в каждом модуле программы) должны быть уникальными. Следовательно, вставив в программу несколько фрагментов, аналогичных приведенному выше, названия меток необходимо изменить.

Оператор переключателя в языке С имеет вид

switch (*УправляющаяПеременная*) {
case *Конст1*: *Операторы1*
case *Конст2*: *Операторы2*
. . .
default: *Операторы3* }

Напомним, что в случае равенства управляющей переменной какой-либо из констант начинают выполняться все операторы от этой метки до конца переключателя. Среди операторов может стоять оператор **break**, и тогда происходит выход из переключателя. Блок **default** не обязательно должен стоять последним. Блок **default** может отсутствовать.

Для конкретности реализуем переключатель

switch (*a*) {
case *0*: *Операторы0*
case *1*: *Операторы1*
case 2: *Операторы2*
default: *ОператорыD* }

Таким образом, если значение переменной *a* равно, например, 1, должны выполниться ***Операторы1*** и ***Операторы2*** и ***ОператорыD*** .

Конечно, эту конструкцию можно реализовать сравнениями и условными переходами, но наиболее красиво и эффективно это можно сделать при помощи так называемой таблицы переходов.

Сначала отработаем вариант **default**, когда *a*>2. Теперь *a* может принимать только значения 0,1 и 2.

Идея переключения состоит в том, что мы введем массив **Table** из трех слов и инициализируем его метками переходов на соответствующие варианты **L0**, **L1**, **L2**. Значения меток будут содержаться в трех рядом стоящих словах. Смещение какого-либо из этих слов можно получить, сложив смещение Table с удвоенным значением переменной *a* (поскольку смещения двух рядом стоящих слов отличаются на 2).

```
        cmp     a,2                 ;Если a>2
        ja      DFLT                ;Вариант default
        mov     BX,a
        shl     BX,1                ;B=2*a
        jmp     CS:Table[BX]
Table   dw      L0,L1,L2
L0:     Операторы0
L1:     Операторы1
L2:     Операторы2
DFLT:   ОператорыD
```

Кстати этот фрагмент может служить примером определения данных в сегменте команд, поэтому необходимо, во-первых, следить, чтобы на них не попало управление, и, во-вторых, адресовать их с явным префиксом CS.

Такой вариант реализации переключателя, очевидно, не ведет к его разрастанию при увеличении числа вариантов.

Переключатель Паскаля **case** реализуется аналогично.

В заключение параграфа рассмотрим простую задачу на условные переходы: с клавиатуры вводятся три числа. Программа должна определить, можно ли из отрезков с такими длинами составить треугольник.

```
        include IO.ASM
```

```
Check macro x,y,z
      mov   AX,x
      add   AX,y
      cmp   z,AX
      ja    No
      endm
      .model  small
      .stack
      .data
x     dw    3 dup (?)                  ;Массив
Inv   db    'Next: $'                  ;Приглашение
      .code
Entry:  mov AX,@data
      mov   DS,AX
      mov   CX,3                       ;Число повторений
      xor   SI,SI                      ;Обнуление SI
Input:  lea DX, Inv
      outstr                           ;Макрос из IO.ASM
      inint x[SI]                      ;Макрос из IO.ASM
      add   SI,2                       ;Наращиваем индекс
      loop  Input                      ;Повторить 3 раза
      Check x,x+2,x+4
      Check x,x+4,x+2
      Check x+4,x+2,x
      outch 'Y'                        ;Макрос из IO.ASM
      jmp   Fin
No:   outch 'N'                        ;Макрос из IO.ASM
Fin:  finish                           ;Макрос из IO.ASM
      end   Entry
```

О пакете макросов IO.ASM см. в приложении.

3.2.3. Циклы

Команды Ассемблера LOOP, LOOPZ, LOOPNZ являются реализациями циклов языков высокого уровня **for** (т. е. циклов с фиксированным числом повторений).

Рассмотрим циклы с предусловием
while (*УсловиеПродолжения*) {*Операторы*} (Язык С)

Их реализацию можно представить следующей схемой:

```
        (Команды проверки условия продолжения)
RPT:    jxxx    AWAY  ; Условие продолжения не выполнено
        Операторы
        jmp     RPT
AWAY:   nop
```

Рассмотрим циклы с постусловием

do ***Операторы*** **while (*УсловиеПродолжения*)** (Язык С)

repeat ***Операторы*** **until** ***УсловиеПродолжения*** (Язык Паскаль)

Их реализацию можно представить следующей схемой:

```
RPT:    Операторы
        (Команды проверки условия продолжения)
        jxxx    RPT     ; Условие продолжения выполнено
```

Цикл с постусловием предпочтительнее цикла с предусловием, поскольку он проще, и условие продолжения цикла проверяется не всегда.

Для примера рассмотрим цикл задержки до нажатия клавиши, аналогичный циклу

while(!kbhit( )); (язык С)

repeat until KeyPressed (язык Паскаль)

Его можно реализовать следующим образом:

```
@@:     mov     AH,0Bh
        int     21h         ;Буфер клавиатуры пуст при AL=0
        test    AL,0FFh     ; Буфер пуст?
        jz      @B
```

На практике пользуются более простой задержкой при помощи функции 08*h* прерывания 21*h*, обеспечивающей ввод символа с клавиатуры без повторения на экране (без эха). Такая задержка полностью эквивалентна оператору **readln** (без параметров) в Паскале и оператору **getch();** в С. Такая задержка пишется короче и, главное, не требует предварительного очищения буфера клавиатуры:

```
        mov  AH,08h
        int  21h            ;Задержка до нажатия клавиши
```

Напишем цикл задержки на заданное число секунд (полагаем, что число секунд, меньшее 60, задано в регистре BL)

```
        mov  AH,2Ch
```

```
int     21h  ;В регистреDH секунды
        mov  BH,DH                ;BH= начальное время
RPT:    int  21h
        sub  DH,BH      ;DH=(начальное - текущее) время
        cmp  BL,DH                 ;Время не истекло?
        jae  RPT
```

3.2.4. Программы с параметрами

Часто необходимо из командной строки *DOS* передать параметры в программу, т. е. выполнить команду *DOS* типа

Имя*.EXE *Параметры.

Чтобы разработать программу, способную воспринимать параметры, необходимо иметь в виду, что строка параметров записывается в *PSP*, начиная с байта 81*h*, а ее длина — в байт *80h*. К этой строке легко получить доступ, помня, что перед началом программы регистр *ES* настроен на начало *PSP* (а для *COM*-программы — и все остальные сегментные регистры).

Следует отметить, что в число символов входят не видимые на экране пробелы (в том числе и пробелы от имени программы до параметров) и символ «возврат каретки» в конце строки.

Приведем программу, отображающую свою строку параметров.

```
       .model  tiny
       .code
       org    100h
Entr:  mov    BL,ES:[80h]  ;BL=Длина строки параметров
       mov    byte ptr ES:[BX+81h],'$'
       mov    AH,09h
       mov    DX,81h       ;DX=Начало строки параметров
       int    21h
       ret
       end Entr
```

4. Работа с внешними устройствами

4.1. Видеопамять и работа с экраном

Напомним, что *текстовым* режимом работы компьютера является режим, при котором в некоторую позицию на экране может быть отображен любой символ из набора 256 символов Каждый символ имеет свои *атрибуты*: цвет символа, цвет фона и признак мерцания.

Графическим режимом работы компьютера является режим, при котором в некоторую позицию на экране может быть отображена точка, называемая *пикселем*. Каждый пиксель имеет свой цвет. Каждый цвет представляет собой смесь красного, зеленого и синего цветов в определенной пропорции. Действующий в данный момент набор цветов называется *палитрой*.

Подчеркнем, что для графического режима решающее значение имеет **быстрота** вывода изображения на экран (например, для анимации). В текстовом режиме быстрота вывода изображения на экран существенного значения не имеет.

Работа с экраном может производиться тремя способами:

- при помощи средств DOS (функции 21-го прерывания),
- при помощи средств BIOS (функции 10-го прерывания),
- при помощи непосредственного обращения к видеопамяти.

Кратко можно охарактеризовать средства DOS как бедные и медленные, средства BIOS как более богатые, но также медленные, работу с видеопамятью как самую быструю и универсальную по результатам, но в то же время довольно неудобную и опасную для операционной системы (при разработке программы придется часто перезагружать компьютер).

4.1.1. Текстовый режим

Разные текстовые режимы различаются числом строк на экране, числом символов в строке и количеством цветов. Мы будем рассматривать только текстовый режим 3, являющийся стандартным для DOS+VGA. Это режим 80×25 с палитрой 16 цветов. Если он почему-либо в данный момент не действует, его можно установить при помощи функции 00 прерывания BIOS 10*h* (0 в регистр AH , 3 в регистр AL).

При выводе на экран в текстовом режиме необходимо иметь в виду следующее. Предполагается, что имеются два виртуальных устройства STDOUT и STDERR, причем оба они соответствуют одному физическому устройству — экрану монитора. Различие между ними состоит в следующем. Допустим, что в файле ABC.EXE имеется программа, в которой имеется вывод как на устройство STDOUT, так и на устройство STDERR. Будучи запущенной обычным образом, эта программа, как обычно, произведет весь вывод на экран. Однако имеется возможность перенаправить вывод программы в файл, т. е. запустить ее из командной строки DOS, например, при помощи команды ABC.EXE > ABC.XXX. В этом случае в текущем каталоге на диске возникнет файл ABC.XXX, в котором будет весь вывод на устройство STDOUT, а на экран этот вывод не поступит. Весь вывод на устройство STDERR по-прежнему будет осуществляться только на экран. Итак, различие между STDOUT и STDERR состоит в том, что вывод на первое устройство может быть перенаправлен в файл, а вывод на второе устройство всегда направляется на экран.

В каждый момент работы компьютера в его распоряжении имеется набор из 256 символов. Все символы занумерованы от 0 до 255 и эти номера называются ASCII-кодами символов (ASCII=American Standard Code for Information Interchange). Хотя все (точнее, почти все) символы имеют графическое изображение, в символьных и строчных константах программы мы можем написать, естественно, только те символы, для которых на клавиатуре есть соответствующие клавиши. Символы, отсутствующие на клавиатуре, приходится указывать их ASCII-кодами.

Таким образом, следующие три строки означают одно и то же:

```
Jura  db    'Юра'
Jura  db    158,0E0h,160
Jura  db    9Eh,224,0A0h
```

Символы, отсутствующие на клавиатуре, приходится указывать их ASCII-кодами. Например, в наборе символов есть символ «пика» (карточная масть) с ASCII-кодом 6. Можно определить:

```
spade db    6
```

Теперь если вывести на экран переменную **spade** как символ, на экран будет выведен символ «пика».

Однако 4 символа из набора могут иметь специальное значение (а могут и не иметь этого значения). Их ASCII-коды:

07*h* — звуковой сигнал
08*h* — перевод курсора на одну позицию влево
09*h* — табуляция
0A*h* — переход на новую строку (на ту же позицию по вертикали)
0D*h* — перевод курсора на начало строки («перевод каретки»)

Одновременно эти же символы имеют и графическое изображение (например, символ 0D*h* графически изображается как музыкальная нота).

Дело в том, что некоторые функции прерываний DOS и BIOS при выводе обрабатывают эти символы указанным специальным образом, а другие функции обрабатывают их так же, как и все другие символы.

Например, применявшаяся нами в программе вывода сообщения на экран функция 09*h* прерывания DOS 21*h* обрабатывает указанные символы специальным образом. Поэтому если мы заменим в программе раздела 2.3 строку

```
Msg            db  "It works$"
```

на фрагмент

```
Msg db      80*12 dup(' ')
   db  35 dup(' '),201,9 dup(205),187,10,13
   db  35 dup (' '),186,"It works!",186,10,13
   db  35 dup (' '),200,9 dup(205),188,10,13
   db  80*12 dup(' ')
   db  7                                    ;Звуковой сигнал
   db  '$'
```

то получим вывод строки **It works!**, обведенной двойной рамкой посередине чистого экрана, и вывод будет сопровождаться звуковым сигналом. (Здесь символы с ASCII-кодами 201, 205, 187, 186, 200 и 188 — символы псевдографики для рисования рамки, 10=0A*h*, 13=0D*h*, пробелы выводятся с целью «кустарной» очистки экрана.)

Аналогично функции 09*h* функция 02*h* прерывания 21*h* выводит на устройство STDOUT один символ, ASCII-код которого занесен в DL. Она также обрабатывает управляющие символы. Напишем при помощи этой функции программу, выводящую на экран подряд все 256 символов набора (приведем только существенную часть программы).

```
        mov  CX,256          ;Число выводимых символов
        mov  AH,02h          ;Номер функции=2
        xor  DL,DL           ;ASCII-код 1-го символа — ноль
RPT:    int  21h             ;Вывод очередного символа
        inc  DL              ;ASCII-код следующего символа
        loop  RPT            ;Повторить 256 раз
```

Будет ли правильно работать такая программа? Нет, она не выведет на экран все 256 символов. Действительно, вместо вывода символа 07*h* она издаст звуковой сигнал, вместо вывода символа 08*h* переведет курсор на одну позицию влево, а потом забьет ранее выведенный в эту позицию символ и т. д. Правда, подавляющее число символов она выведет на экран правильно.

Функция 06*h* прерывания 21*h* отличается от функции 02*h* только тем, что не обрабатывает особым образом указанные спецсимволы. Поэтому если в приведенной выше программе заменить строку

```
        mov    AH,02h                ;Номер функции=2
```

на строку

```
        mov    AH,06h                ;Номер функции=6
```

программа корректно выведет на экран все 256 символов символьного набора.

Функция 40*h* прерывания 21*h* так же, как и функция 09*h*, выводит строку по адресу DS:DX, но фиксируется не конец строки, а ее длина, которую необходимо поместить в CX. Кроме того, если BX=1, то вывод происходит на устройство STDOUT, если BX=2, то вывод происходит на устройство STDERR (при других значениях BX происходит запись строки в файл).

Не будем подробно рассматривать функции прерывания BIOS 10*h*, скажем только, что они дают возможность позиционировать курсор (т. е. выводить символы в указанное место на экране), задавать цвет выводимых символов и фона (включая режимы мигания), а также, наоборот, считывать текущее положение курсора, код и атрибуты символа в заданном месте экрана.

Рассмотрим непосредственную работу с видеопамятью в текстовом режиме. Видеопамять представляет собой память на видеокарте, встроенную в адресное пространство процессора по адресу B800*h*:0000*h* (для текстового режима). На экранный текстовый буфер отводится 32 К. Конечно, физически на видео-

карте памяти обычно намного больше, но компьютер «видит» ее через «щель» ширины 32 К. При необходимости эта «щель» может передвигаться по физической видеопамяти видеокарты, впрочем в текстовом режиме такой необходимости не возникает.

Каждой из 80×25 позиций на экране последовательно отводятся два байта видеопамяти (младший байт для ASCII-кода символа, старший байт для его атрибутов). При занесении некоторого двоичного кода в такую пару байт, на экране немедленно отображается соответствующий символ. Поэтому вывод на экран можно производить непосредственно из программы пересылкой данных в видеопамять без помощи DOS и BIOS.

При этом необходимо учитывать соответствие байтов видеопамяти и позиций на экране. Пусть m=0.1,...,79 — номер символа в строке, n=0,1,...24 — номер строки. Тогда смещение относительно начала видеопамяти байта ASCII-кода m-го символа в n-й строке будет $2*m+80*2*n$, а смещение байта атрибута этого символа будет $2*m+80*2*n+1$.

Следующая программа забивает среднюю строку экрана мерцающими белыми буквами А на голубом фоне.

```
        .model small
        .code
row     equ 13
prg:    mov    AX,0B800h
        mov    ES,AX        ;Настроили ES на видеопамять
        mov BX,80*2*row     ;BX=смещ. 1-го симв. 13-й строки
        xor    SI,SI
        mov    CX,80
RPT:    mov    ES:[BX+SI],byte ptr 41h      ;Код символа
        mov ES:[BX+SI+1],byte ptr 9Fh      ;Атрибут симв.
        add    SI,2
        loop   RPT
        mov    AH,4Ch
        int    21h
        .stack  256
        end  prg
```

Несколько замечаний к этой программе. Если вместо строк

```
RPT:  mov   ES:[BX+SI],byte ptr 41h
      mov   ES:[BX+SI+1],byte ptr 9Fh
```

написать просто

```
RPT:  mov   ES:[BX+SI],41h
      mov   ES:[BX+SI+1],9Fh
```

то компилятор будет не в состоянии определить, заносим ли мы константы в байты или слова. По умолчанию он примет, что константы заносятся в байты и в данном случае «угадает». В результате программа будет работать правильно, но при компиляции будет выдано предупреждение: Argument needs type override (необходимо явно определить тип операнда).

Вместо этих двух строк можно написать одну, сразу занося в видеопамять константу размером в слово, но при этом необходимо учесть «перевернутость» байтов слова в памяти (старший байт по старшему адресу):

```
RPT:  mov   ES:[BX+SI],9F41h
```

Здесь уже не нужно явно определять тип операнда, поскольку константа 9F41*h* имеет размер слова.

4.1.2. Графический режим

Основная проблема графического режима при работе компьютера в реальном режиме состоит в том, что области адресного пространства 64 К, выделенной для графического буфера, недостаточно для отображения всего экрана. Существует очень большое число видеорежимов и разновидностей видеокарт, по-разному решающих эту проблему.

Рассмотрим лишь один наиболее простой и употребительный графический режим VGA 320×200, 256 цветов. В нем каждый пиксель описывается одним байтом, значение которого определяет цвет пикселя.

Переход в графический режим VGA 320×200, 256 цветов осуществляется при помощи функции 00*h* прерывания BIOS 10*h* установкой режима 13*h*.

Приведем программу, которая в графическом режиме 13*h* рисует две прямые линии: одну красную горизонтальную, другую

зеленую под углом 45°. Эта программа демонстрирует отображение пикселей как с помощью BIOS (функция 0*Ch* прерывания 10*h*), так и с помощью непосредственного обращения к видеопамяти. Второй способ действует гораздо быстрее первого.

Рассмотрим соответствие байтов видеопамяти и позиций пикселей на экране. Пусть m=0,1,...,319 — номер пикселя в строке, n=0,1,...200 — номер строки пикселей. Тогда смещение относительно начала видеопамяти байта, соответствующего m-му пикселю в n-й строке будет m+320*n.

```
putpix  macro  x,y,color      ;Пиксель при помощи BIOS
        mov    CX,x
        mov    DX,y
        mov    AL,color
        mov    AH,0Ch
        int    10h
        endm
        .model  small
        .code
prg:    mov    AX,0A000h
        mov    ES,AX              ;Настройка ES на видеопамять
        mov    AX,0013h
        int    10h                ;В графический режим 13h
        mov    CX,200             ;Линии в 200 пикселей
        xor    SI,SI
RPT:    push   CX
        putpix SI,SI,2            ;При помощи BIOS
        mov    ES:[SI]+32000,byte ptr 4  ;В видеопамять
        inc    SI
        pop    CX
        loop   RPT                ;Повторить 200 раз
        mov    AH,08h
        int    21h                ;Задержка до нажатия клавиши
        mov    AX,0003h
        int    10h                ;В текстовый режим 03h
        mov    AH,4Ch
        int    21h                ;Возврат в DOS
        .stack 256
        end    prg
```

Конечно, в приведенной программе скорость выполнения значения не имеет, однако следует отметить, что вывод на экран при помощи функций *BIOS* приблизительно в 10-20 раз медленнее, чем прямое копирование в видеопамять.

В видеорежимах *VGA* и *SVGA* с более высоким разрешением, чем 320×200, размера видеобуфера уже не достаточно для отображения всего экрана, поэтому приходится отображать изображение по очереди на части экрана.

4.2. Клавиатура

Клавиатура является физическим устройством, обычно назначаемым виртуальному устройству стандартного ввода *STDIN* (поэтому можно перенаправить ввод клавиатуры на ввод из файла или из другой программы).

На клавиатуре имеются клавиши двух разновидностей: алфавитно-цифровые и управляющие. При нажатии любой клавиши в кольцевой буфер ввода *BIOS* размером 16 слов поступает двухбайтный код. Если нажата алфавитно-цифровая клавиша, в младший байт заносится *ASCII*-код введенного символа, а в старший — *скан-код* нажатой клавиши. Скан-код полностью идентифицирует нажатую клавишу.

Если же нажата управляющая клавиша, в младший байт записывается ноль. Этот нулевой байт в совокупности со скан-кодом управляющей клавиши называется ее *расширенным ASCII-кодом.* Алфавитно-цифровая клавиша, нажатая совместно с клавишей *Alt*, также рассматривается как управляющая, и она также имеет расширенный *ASCII*-код (с нулевым младшим байтом).

Эти коды вводятся в буфер и заполняют его по мере нажатия на клавиши независимо от работы программы. Буфер может заполниться полностью, и тогда ввод с клавиатуры станет невозможным. Буфер может также быть пустым.

Если в программе имеется ввод из *STDIN*, она считывает из буфера необходимое число символов (двухбайтных кодов), освобождая место в буфере для новых символов, которые могут поступить от клавиатуры. Если на момент ввода буфер пуст, программа «зависает» в режиме непрерывного опроса буфера.

Ввод символа с клавиатуры может осуществляться с сопровождением его отображения на экране (с эхом) или без него и с реакцией на нажатие сочетания клавиш *Ctrl-Break* или без нее. Эти особенности целиком определяются программой, т. е. примененными в ней для ввода прерываниями.

Функции ввода *DOS* прерывания 21*h* осуществляют только ввод в программу *ASCII*-кода символа, например:

```
mov     AH,08h          ;Без эха, с реакцией на Ctrl-Break
int     21h             ;В AL ожидается ASCII-код символа
```

Однако если можно ожидать не только нажатия алфавитно-цифровой клавиши, но и управляющей, а также необходимо определить, какая управляющая клавиша была нажата, необходимо применить более сложную конструкцию. При этом необходимо учесть, что однократное считывание нажатой управляющей клавиши даст ноль (записанный в буфер вместо *ASCII*-кода).

```
mov     AH,08h
int     21h
cmp     AL,0              ;Символ или управляющая клавиша?
jne     SymbolKey
int     21h         ;Повторное считывание при управл. клавише
jmp   short  ControlKey
...
SymbolKey:                      ;В регистре AL ASCII-код символа
...
ControlKey:     ;В регистре AL скан-код управляющей клавиши
```

Функция ввода *06h* прерывания 21*h* в отличие от других функций ввода *DOS* и *BIOS* не приостанавливает выполнение программы. Она считывает символ из буфера, если последний не пуст, и во всех случаях выполнение программы продолжается.

Функция *0Bh* прерывания 21*h* возвращает в *AL* ноль, если буфер ввода пуст и 0*FFh*, если он не пуст.

Функция *0Ch* прерывания *21h* очищает буфер и вызывает какую-либо другую функцию ввода прерывания *21h* (указывается в *AL*) для считывания символа.

По сравнению со средствами ввода *DOS*, средства ввода *BIOS* (прерывание *16h*) предоставляют большие возможности по анализу состояния клавиатуры.

У прерывания *BIOS* имеются тройки функций (например, *00h*, *10h* и *20h*) соответственно для 84-клавишной, 102-клавишной и 122-клавишной клавиатур.

Кратко перечислим возможности прерывания *16h BIOS*, связанные с клавиатурой:

- одновременное считывание *ASCII*-кодов и скан-кодов или расширенных *ASCII*-кодов управляющих клавиш *(00h, 10h, 20h)*,
- одновременное считывание *ASCII*-кодов и скан-кодов или расширенных *ASCII*-кодов управляющих клавиш без удаления из буфера ввода *(01h, 11h, 21h)*,
- определение состояния клавиатуры (переключателей, светодиодов и проч.) *(02h, 12h, 22h)*,
- помещение символа в буфер ввода из программы (*05h*),
- определение типа клавиатуры (*09h*).

В принципе, как и при работе с видеопамятью, можно осуществлять прямой доступ к буферу ввода(по адресам *0:41Eh–0:43Dh*, указатели начала и конца буфера в *0:41Ah* и *0:41Ch)*, однако это не дает существенного выигрыша в быстродействии по сравнению со средствами *DOS* и *BIOS*, и поэтому практически не применяется.

4.3. Мышь

Работа с мышью производится при помощи специальной резидентно установленной программы — драйвера мыши (обычно в файле *MOUSE.COM* или *MOUSE.SYS*) и прерывания *33h*. Номера функций этого прерывания заносятся в регистр *AX*. Общее чис-

ло функций зависит от применяемого драйвера и достигает 50. Главные из них следующие.

Функция 0*h* — инициализация мыши. В регистре *AX* возвращается 0, если мыши или ее драйвера нет, и значение *FFFFh* при удачной инициализации. В регистре *BX* возвращается число клавиш мыши.

Функция 01*h* — Показать указатель мыши.

Функция 02*h* — Спрятать указатель мыши.

Функция 03*h* — Состояние мыши. В трех младших битах *BX* возвращается состояние клавиш (бит 0 — левая, бит 1 — правая, бит 2 — средняя). В регистрах CX и DX возвращаются соответственно координаты *x* и *y* указателя мыши (в некоторых видеорежимах эти возвращаемые значения необходимо разделить на 2, 4 или 8).

Функция *03h*, в принципе, дает возможность полностью обрабатывать мышь в программе, однако это неудобно, поскольку приходится создавать циклы непрерывного опроса состояния мыши. Более удобна следующая функция:

Функция 0C*h* — Установка и сброс обработчика событий мыши. В паре *ES:DX* задается адрес обработчика, представляющий собой *far*-процедуру. В регистре *CX* задается 0 для сброса обработчика или условие вызова, определяемое семью младшими битами (бит 0 — перемещение, биты 1, 3, 5 — нажатие клавиш, биты 2, 4, 6 — отпускание клавиш). На входе в обработчик в *AX* будет условие вызова, в *BX* — состояние клавиш, в *CX* и *DX* — координаты указателя мыши, в *SI* и *DI* — счетчики перемешения указателя по экрану. Перед завершением программы установленный обработчик необходимо сбросить.

4.4. Файлы

Поддержка файловой системы является главнейшей функцией любой операционной системы. Операционная система *DOS* поддерживает организацию файлов на дисках *FAT* (*File Allocation Table*).

Здесь не представляется возможным рассмотреть более или менее подробно нижний уровень работы с дисками. Кратко рассмотрим лишь некоторые средства, предоставляемые *DOS* для работы с файлами изнутри программы на Ассемблере.

Средства *DOS* можно разделить на две группы. В первую группу можно включить средства по реорганизации файловой системы изнутри программы, по поиску файлов и проч. Хотя такие средства имеются, мы также не будем их рассматривать.

Вторую группу составляют средства по открытию файла и чтению из него или записи в него.

4.4.1. Создание, открытие и закрытие файла

Для работы с файлом необходимо сначала сопоставить физический файл на диске с его *дескриптором* в программе. Дескриптор — переменная размером в слово, его значение возвращается функциями создания или открытия файла *3Ch*, 3*Bh* и 3*Dh* прерывания 21*h*.

Поскольку внешние устройства рассматриваются как файлы, имеются 3 стандартных дескриптора:

Ст. дескриптор	Значение	Назначение
STDIN	*0*	Клавиатура
STDOUT	*1*	Экран
STDERR	*2*	Экран

Имя файла на диске, записанное по правилам *DOS*, должно быть определено в так называемой *ASCIIZ*-строке -- в строке, заканчивающейся нулем. При создании файла можно задать его атрибут, обычно его полагают равным 0. При открытии существующего файла необходимо задать режим доступа. Приведем пример создания нового и открытия существующего файла и последующего их закрытия.

```
.data
Path_1  db   'C:\USER\ABC.TXT',0          ;ASCIIZ-строка
Path_2  db   'DEF.DAT', 0                 ;ASCIIZ-строка
Dscr_1  dw   ?                  ;Будущий дескриптор файла
Dscr_2  dw   ?                  ;Будущий дескриптор файла
```

```
...
.code
...
; Создаем файл
        mov  AH,3Ch          ;Номер функции создания
        xor  CX,CX           ;Атрибут=0
        lea  DX,Path_1       ;Адрес ASCIIZ-строки в DX
        int  21h             ;Создание нового файла
        jc   Error1          ;Уход на обработку ошибки
        mov  Dscr_1,AX       ;Сохранение дескриптора
; Открываем файл
        mov  AH,3Dh          ;Номер функции открытия
        mov  AL,0            ;Только на чтение
        lea  DX,Path_2       ; Адрес ASCIIZ-строки в DX
        int  21h             ;Открытие существующ. файла
        jc   Error2          ;Уход на обработку ошибки
        mov  Dscr_2,AX       ;Сохранение дескриптора
        ...
; Закрываем файлы
        mov  AH,3Eh          ;Номер функции закрытия
        mov  BX,Dscr_1
        int  21h             ;Закрытие файла
        mov  BX,Dscr_2
        int  21h             ;Закрытие файла
```

К этому примеру необходимо дать следующие комментарии:

- Максимальное число одновременно открытых файлов определяется параметром директивы *FILES* в системном файле *CONFIG.SYS*.
- При создании файла при помощи функции *3Ch*, если такой файл уже существовал, его длина полагается равной 0 (т. е. его старое содержимое теряется). Чтобы избежать этого, применяется функция *5Bh*, отличающаяся от *3Ch* только тем, что в случае существования файла его содержимое сохраняется.
- В случае успеха при создании/открытии файла флаг *CF* выставляется в *0*, а в *AX* возвращается значение дескриптора.

В случае неудачи при создании/открытии файла флаг *CF* выставляется в *1*, а в *AX* возвращается код ошибки (2 — файл не найден, 3 — путь не найден, 4 — слишком много открытых файлов, 5 — доступ запрещен/нет места, и проч.). Не следует путать значения дескрипторов, совпадающие с кодами ошибок (различаются по сопутствующему значению флага *CF*).

- Режимы открытия файла (передается в *AL* при функции *3Dh*): 0 — только на чтение, 1— только на запись, 2 — на чтение и запись, имеются и другие режимы.
- При закрытии файла все данные сбрасываются из буфера на диск и корректируется таблица *FAT*. Дескриптор файла становится свободным и может быть применен для других целей (например, для связи с другим физическим файлом).
- Имеется функция *5Ah*, работающая так же, как функция *3Bh*, но создающая временный файл, который при закрытии удаляется. Его *ASCIIZ*-строка должна иметь вид:

```
TempFile     db     '\', 13 dup(0)          ;13 нулей
```

4.4.2. Чтение из файла и запись в него

Работа с файлом в программе (чтение и запись) осуществляется при помощи *буфера*, который необходимо предусмотреть в сегменте данных, его адрес задается парой *DS:DX*.

Функция 3F*h* — чтение из файла (или устройства).
Задаются:

AH=3Fh

BX= дескриптор файла

CX= число считываемых байт

DS:DX= адрес буфера-приемника

Возвращаются:

AX= число считанных байт при успехе, код ошибки при неудаче. Флаг *CF=0* при успехе и *CF=1* при неудаче.

Каждое новое чтение при помощи функции *3Fh* начинается не с начала файла, а с того места, на котором закончилось прошлое

чтение/запись (если указатель чтения/записи не был перемещен при помощи функции 42*h*).

Функция 40*h* — запись в файл (или на устройство).
Задаются:

AH=40*h*

BX= дескриптор файла

CX= число записываемых байт

DS:DX= адрес буфера-источника

Возвращаются:

AX= число записанных байт при успехе, код ошибки при неудаче. Флаг *CF=0* при успехе и *CF=1* при неудаче.

Каждая новая запись при помощи функции 40*h* начинается не с начала файла, а с того места, на котором закончилось прошлое чтение/запись (если указатель чтения/записи не был перемещен при помощи функции 42*h*). При задании *CX=0* файл будет «обрезан», т. е. удалены все последующие записи.

Функция 68*h* — сброс буфера в файл.
Задаются:

AH=68*h*

BX= дескриптор файла

DS:DX= адрес буфера-источника

Возвращаются:

AX= код ошибки при неудаче. Флаг *CF=0* при успехе и *CF=1* при неудаче.

Функция 42*h* — установка указателя файла.
Задаются:

AH=42*h*

BX= дескриптор файла

CX:DX= сдвиг указателя (как 32-значное число, может быть отрицательным)

AL= исходное положение (0 — начало файла, 2 — конец файла, 1 — текущая позиция)

Возвращаются:

CX:DX= новое значение указателя при успехе.

AX= код ошибки при неудаче. Флаг *CF=0* при успехе и *CF=1* при неудаче.

Следующая операция чтения/записи будет происходить с установленного значения указателя. При значениях *CX=0*, *DX=0* и *AL=2* эта функция возвращает в *CX:DX* длину файла в байтах.

Функция 41*h* — удаление файла.
Задаются:

AH=41*h*

DS:DX= адрес *ASCIIZ*-строки

Возвращаются:

AX= код ошибки при неудаче. Флаг *CF=0* при успехе и *CF=1* при неудаче.

Функция 36*h* — состояние диска.
Задаются:

AH=36*h*

DL= номер диска (0 — текущий. 1 — A:, 2 — *B*:, *3* — *C*: и т. д.)

Возвращаются:

AX= число секторов на кластер

BX= число свободных кластеров

CX= число байт на сектор

DX= общее число кластеров на диске

5. Резиденты. Перехватчики. Обработка событий

5.1. Резидентные программы

В отличие от *UNIX* и *Windows NT* операционная система *DOS* является однозадачной. Это означает, что все ресурсы процессора отданы одной выполняемой задаче. Однако при выполнении некоторой программы в памяти могут находиться и другие программы, которые в данный момент не выполняются, но на них может быть передано управление при наступлении некоторого события. Таким образом в некотором смысле преодолеваются недостатки однозадачной системы — на макроуровне создается впечатление, что работают сразу несколько программ.

Программа, которая остается в памяти, но не выполняется, а ожидает наступления некоторого события, после которого на нее будет передано управления, называется резидентной или программой *TSR* (*Terminate and Stay Resident in memory* — завершить выполнение программы и оставить ее резидентной в памяти).

Необходимо особо подчеркнуть, что к резидентным программам предъявляются крайне жесткие требования по объему занимаемой памяти (они уменьшают память, доступную транзитным программам), и поэтому они почти всегда разрабатываются на Ассемблере.

Поскольку резидентные программы, резидентная часть которых занимает большой объем памяти, смысла не имеют, они обычно разрабатываются в формате *COM*-файла.

В связи с разработкой резидентных программ возникают следующие проблемы различной важности и сложности:

- Программа должна загрузиться для инсталляции, закончить работу и остаться резидентной в памяти. Для экономии памяти желательно, чтобы в памяти оставалась не вся программа, а лишь ее некоторая «резидентная» часть.
- Должен быть предусмотрен перехватчик событий, передающий управление в резидентную программу при наступлении некоторого события.
- Должен быть обеспечен возврат в исходную программу по окончании работы резидента.

- Желательно наличие возможности выгрузки резидентной программы из памяти. При этом должно быть обеспечено прекращение действия перехватчика событий.
- Желательна блокировка повторной загрузки резидентной программы в память.

Наиболее просто решается первая из перечисленных проблем — при помощи прерывания *27h*. Перед его вызовом в регистр *DX* необходимо загрузить длину резидентной части в байтах (от начала *PSP)*, таким образом часть программы, выполняющая первоначальную загрузку и не нужная для работы резидента, не будет сохранена в памяти. Это прерывание приводит к тому, что операционная система прекращает выполнение программы и помещает ее резидентную часть по наинизшему свободному адресу (можно загрузить резидент и в верхнюю память при помощи команды *DOS LOADHIGH)*.

Прерывание *27h* не позволяет оставить резидентной программу размером более 64 Кбайт, что в большинстве случаев является достаточным. Функция *DOS 31h* прерывания *21h* делает то же самое, но длину резидентной части указывается в 16-байтных параграфах, что дает возможность оставить резидентной программу размером более 64 Кбайт.

Резидентная программа обязательно должна иметь две точки входа: одну при начальной инсталляции, и другую при обращении к резиденту. Чаще всего резидентная часть представляет собой *far*-процедуру.

Приведем типичную структуру резидентной программы.

```
      .model  tiny
      .code
      org  100h
Entry:
      jmp Install
Resid  proc
; КОД РЕЗИДЕНТНОЙ ЧАСТИ
      retf
; ДАННЫЕ РЕЗИДЕНТНОЙ ЧАСТИ
Resid  endp
Install:
```

```
;КОД ИНСТАЛЛЯЦИОННОЙ ЧАСТИ
        mov    DX, offset Install
        int    27h              ;Завершить и оставить в памяти
;ДАННЫЕ ИНСТАЛЛЯЦИОННОЙ ЧАСТИ
        end    Install
```

Дадим некоторые пояснения к приведенной схеме. Резидентная часть оформлена как *far*-процедура и она не выполняется при инсталляции. Выполнение инсталляционной части заканчивается прерыванием *27h*, оставляющим в памяти резидентную часть. Перед вызовом прерывания *27h* необходимо загрузить длину резидентной части от начала *PSP* (в байтах) в *DX*

5.2. Обработчики прерываний

Самой сложной является вторая из перечисленных проблем – организация перехватчика события, передающего управление резиденту. Здесь существует большое число остроумных идей и методов, но наиболее общим подходом является переопределение адресов некоторых векторов прерываний на адрес резидентной части.

Напомним, что начиная с нулевого адреса в памяти располагаются 256 четырехбайтных векторов прерываний, каждый из которых представляет собой некоторый адрес в формате СЕГМЕНТ:СМЕЩЕНИЕ, или 4 нулевых байта. Адрес самого вектора прерывания можно получить, умножив номер прерывания на 4.

Действие команды **int *Номер*** состоит в том, что: 1) в стек заносится содержимое регистра *FLAGS*, 2) в стек заносится дальний адрес команды, следующей за командой *int* (адрес возврата), 3) управление передается в *far*-процедуру, адрес которой содержится в векторе прерывания с номером, указанным в команде *int*, 4) процедура, на которую передано управление, заканчивается командой **iret**, которая восстанавливает содержимое регистра *FLAGS* и передает управление на адрес возврата, т. е. на команду, следующую за командой *int*.

Существенно, что многие векторы прерываний пусты, например векторы *60h–65h*, *78h–8Ch* и другие.

Теперь перед нами два пути: 1) вставить в пустой вектор прерывания адрес своей процедуры-обработчика, 2) заменить в векторе прерывания адрес процедуры, используемый системой, на адрес своей процедуры-обработчика. Применяются оба метода в зависимости от задачи, стоящей перед программистом.

Чтобы понять, какой из этих методов применяется в каждом конкретном случае, рассмотрим природу события, которое мы хотим перехватить и обработать. Где это событие может произойти? Событие может произойти: 1) в некоторой программе пользователя (например, не найден файл, который пытались открыть в программе), 2) в процессоре при возникновении некоторой исключительной ситуации (например, деление на ноль), 3) во внешнем устройстве (например, нажата некоторая клавиша на клавиатуре). Обратим внимание на то, что для случаев 2) и 3) системой уже предусмотрены соответствующие прерывания.

Очевидно, что в первом случае следует записать в пустой вектор прерывания адрес нашего нового прерывания, а во втором и третьем случаях заменить в соответствующем векторе прерывания адрес системной процедуры на процедуру нашего обработчика.

Обработчик должен быть оформлен следующим образом:

```
MyHandler   proc  far
        КОД НОВОГО ОБРАБОТЧИКА
            iret
MyHandler   endp
```

Команда **iret** эквивалентна паре команд **popf** и **retf**, т. е. восстанавливает регистр флагов и осуществляет возврат по адресу, занесенному в стек командой **int**.

Если мы хотим установить наш обработчик в пустой вектор прерывания, мы можем непосредственно вписать полный адрес нашего обработчика в компоненты пустого вектора прерывания, но удобнее сделать это при помощи специальной функции 25*h* прерывания 21*h*:

Функция 25*h* — заполнить вектор прерывания.
Задаются:
AH=25*h*

AL= Номер вектора, в который устанавливается прерывание
DS=Сегментная часть адреса процедуры нового прерывания
DX=Смещение адреса процедуры нового прерывания

В нашем примере это запишется следующим образом (в качестве пустого вектора прерывания выбираем 60*h*):

```
mov   AX,2560h
push  seg  MyHandler
pop   DS
mov   DX, offset  MyHandler
int   21h
```

Теперь необходимо только оставить резидентной в памяти процедуру *MyHandler*, как это описано выше, и после этого любым программам будет доступно прерывание 60*h* с новым обработчиком.

Хотя установка нового прерывания будет действовать только до первой перезагрузки компьютера, вообще говоря, полагается после использования прерывания восстановить старое.

Для сохранения старого прерывания имеется специальная функция *DOS* 35*h*.

Функция 35*h* — считать вектор прерывания.
Задаются:
AH=23*h*
AL= Номер считываемого вектора прерывания
Возвращаются:
ES= сегментная часть адреса процедуры прерывания
BX= смещение адреса процедуры прерывания

Итак общий сценарий установки нового и восстановления старого прерывания следующая:

1. Запомнить старое значение вектора прерывания в двух словах при помощи функции 35*h*,
2. Установить новое прерывание при помощи функции 25*h*,
3. Использовать новое прерывание,
4. Установить старое прерывание при помощи функции 25*h*.

На практике вместо этапов 1 и 4 можно просто перезагружать компьютер.

В принципе, тот же самый сценарий используется и при переопределении прерываний от внешних устройств и от процессора. Однако в этом случае мы должны переопределять не пустой вектор, а вектор, на который «повешен» системный обработчик прерывания. Системные обработчики прерываний от внешних устройств выполняют много сложных задач и сомнительно, чтобы мы смогли написать свой обработчик, который выполнял бы все эти задачи и еще некоторые дополнительные. Было бы желательно лишь дополнить системный обработчик внешних прерываний некоторыми новыми особенностями.

Для этого можно в начале или в конце нового обработчика вызвать системный обработчик.

Пусть в нашем примере мы сохранили старый вектор прерывания в переменной типа двойного слова **SystHandler** при помощи функции 31*h*. Тогда процедура нового обработчика примет вид:

```
MyHandler   proc   far
        pushf
        call   SystHandler ;Вызов системного обработчика
        КОД  НОВОГО ОБРАБОТЧИКА
               iret
MyHandler   endp
```

или

```
MyHandler   proc   far
        КОД  НОВОГО ОБРАБОТЧИКА
               pushf
        call   SystHandler ;Вызов системного обработчика
               iret
MyHandler   endp
```

В первом случае сначала отрабатывает системный обработчик, а затем новый обработчик, вносящий в обработку прерывания от внешнего устройства некоторые дополнительные особенности, во втором случае сначала отрабатывает новый обработчик, а затем системный.

Наиболее часто переопределяются следующие прерывания от внешних устройств:
Системный таймер — прерывание 08*h* *(IRQ 0)*
Клавиатура – прерывание 09*h* *(IRQ 1)*

Причины переопределения прерывания клавиатуры 09*h* понятны: это простейший способ заставить отработать резидентную программу (например, нажатием клавиши *Scroll Lock* переключить клавиатуру с английского языка на русский при загруженном резиденте-русификаторе). Конечно, при переопределении прерывания клавиатуры необходимо позаботиться о том, чтобы выполнялся и системный обработчик этого прерывания.

Приведем пример простой, но тем не менее имеющей некоторое практическое значение резидентной программы. При наборе некоторого текста, в котором наряду с русскими встречаются и английские слова, наборщик, не глядя на экран, иногда забывает переключить клавиатуру с английского на русский, таким образом несколько фраз оказываются набранными не на том регистре и их необходимо перепечатывать. Во избежание этого хотелось бы сопроводить нажатие клавиши на одном из регистров (русском или английском) звуковым сигналом.

Для определенности предположим, что клавиатура переключается нажатием на клавишу *Scroll Lock*, хотя это не принципиально — можно было бы задаться любой другой клавишей или комбинацией клавиш. Напишем резидентную программу, после загрузки которой при работе в любой другой программе или в самой операционной системе при одном из положений переключателя *Scroll Lock* нажатие любой клавиши будет сопровождаться коротким звуковым сигналом.

Конечно, обработку *Scroll Lock* и генерацию звукового сигнала можно заменить на любую другую дополнительную обработку прерывания 09*h* от нажатия клавиш на клавиатуре.

Идея программы состоит в том, что мы заменим адрес стандартного обработчика прерывания от клавиатуры, записанный в векторе *09h*, на адрес нашего обработчика, и при поступлении прерывания от клавиатуры будет работать именно наш обработчик. Конечно, мы не сможем полностью промоделировать работу системного обработчика прерываний от клавиатуры —

он слишком сложен, да нам это и не нужно. Нам нужно лишь дополнить его действия. Поэтому в нашем обработчике мы сначала вызовем системный обработчик (адрес которого мы предусмотрительно сохранили), а затем выполним необходимые дополнительные действия. (Можно было бы выбрать обратную последовательность: сначала выполняются дополнительные действия, а затем прорабатывает системный обработчик.)

Сначала резидентная программа запускается для инсталляции, а затем при каждом перехвате прерывания работает только резидент. Поэтому резидентная программа должна иметь инсталляционную и резидентную части, а также инсталляционный и резидентный входы.

Основное требование к резидентным программам — минимальный объем занимаемой памяти. Поэтому они чаще всего разрабатываются в *COM*-формате, и устроены так, что в памяти остается только резидентная часть, т. е. программа после инсталляции фактически уничтожает свою инсталляционную часть!

Кроме того, опять же из соображений экономии памяти резидентная программа должна блокировать свою повторную инсталляцию, чтобы в памяти не находились две или более копии одной и той же программы.

```
        .model  tiny
        .code
        org  100h
;НАЧАЛО РЕЗИДЕНТНОЙ ЧАСТИ
InstallEntry:
         jmp  Install                          ;Обход резидента
NewHandler  proc
         pushf
         call  CS:OldHandler       ;Вызов сист. обработчика
         call  MySound           ;Дополнительные действия
         iret
OldHandler   dd   ?      ;4 байта для системного обработчика
include   MYSOUND.ASM          ;Включение текста процедуры
NewHandler   endp
;КОНЕЦ РЕЗИДЕНТНОЙ ЧАСТИ
```

```
;НАЧАЛО ИНСТАЛЛЯЦИОННОЙ ЧАСТИ
Install:          ;Проверка на попытку повторной инсталляции
mov     AX,3509h
int     21h                   ;Запрос действующего обработчика
cmp   BX,offset NewHandler  ;Действует ли новое прерыв.?
jne     Cont                                  ;Не действует
mov     AH,09h                                  ;Действует
lea     DX,Reinst
int     21h    ;Сообщение о попытке повторной инсталляции
jmp     NoInst
Cont:                         ;Начало фактической инсталляции
mov     AX,3509h                  ;Прерывание от клавиатуры
int     21h                       ;Запрос старого обработчика
mov     word ptr OldHandler,BX      ;Смещ. старого прер.
mov     word ptr OldHandler+2,ES    ;Сегм. старого прер.
mov     AX,2509h
mov     DX,offset NewHandler      ;Смещение нового прер.
int     21h                       ;Новый вектор прерывания 09h
mov     AH,09h
lea     DX,Installed
int     21h
mov     DX,offset Install         ;Длина резидентной части
int     27h                       ;Завершить и оставить в памяти
NoInst:  ret          ;Обычное завершение COM-программы
Reinst db  10,13,'BEEP is already installed!'
       db  10,13,7,'$'
Installed  db  10,13
       db  'BEEP is successfully installed'
       db  10,13,7,'$'
end   InstallEntry
;КОНЕЦ ИНСТАЛЛЯЦИОННОЙ ЧАСТИ
```

Не вдаваясь в подробности, приведем текст процедуры, генерирующей звуковой сигнал при одном из двух положений клавиши *Scroll Lock*.

```
MySound  proc
push    AX
push    CX
push    DX
mov     AH,12h                    ;Клавиатура 101/102 клавиши
int     16h                             ;Состояние клавиатуры
```

```
and     AL,00010000b                    ;Байт 4=Scroll Lock
jz      EndSnd
mov     AL,10110110b
out     43h,AL
mov     AL,0Dh          ;Делитель частоты (младшие разряды)
out     42h,AL
mov     AL,6h           ;Делитель частоты (старшие разряды)
out     42h,AL
in      AL,61h
or      AL,03h
out     61h,AL
mov     CX,0000h
mov     DX,05120h            ;Длительность (старшие разряды)
mov     AH,86h               ;Длительность (младшие разряды)
int     15h                                         ;Задержка
in      AL,61h
and     AL,11111100b
out     61h,AL
EndSnd:   pop    DX
        pop    CX
        pop    AX
        ret
MySound  endp
```

Размер исполняемого файла с этой программой составляет всего 191 байт.

Если бы мы захотели, чтобы наши дополнительные действия выполнялись до отработки системного обработчика, недостаточно было бы переставить их местами (за счет различий в командах **iret** и **ret**). Можно было бы написать в этом случае:

```
NewHandler  proc
        call MySound              ;Дополнительные действия
        pushf
        jmp CS:OldHandler          ;Вызов сист. обработчика
        iret
NewHandler  endp
```

В этом случае возврата в новый обработчик уже не происходит, возврат в место вызова осуществляется непосредственно из системного обработчика.

В самом общем случае, когда некоторые дополнительные действия надо выполнить до отработки системного обработчика, а некоторые -- после, новый обработчик будет иметь вид

```
NewHandler  proc
;ДОПОЛНИТЕЛЬНЫЕ ДЕЙСТВИЯ ДО СИСТЕМНОГО
;ОБРАБОТЧИКА
        pushf
  call  CS:OldHandler              ;Вызов сист. обработчика
;ДОПОЛНИТЕЛЬНЫЕ ДЕЙСТВИЯ ПОСЛЕ СИСТЕМНОГО
;ОБРАБОТЧИКА
        iret
NewHandler  endp
```

В *дополнительные действия до системного обработчика* может, конечно, входить анализ текущей ситуации и (в некоторых вариантах) обход вызова системного обработчика

Прерывание системного таймера *08h* генерируется постоянно с частотой 18,2 Гц. Переопределив это прерывание, можно добиться того, чтобы пользователю казалось, что некоторые действия выполняются постоянно (на самом деле они будут происходить 18 раз в секунду, но человек будет воспринимать их как постоянные — например, вывод на экран текущего времени).

Поскольку использование прерывания от системного таймера очень популярно, предусмотрено специальное прерывание-«заглушка» **1С*h***, обработчик которого состоит только из одной команды **iret**, и которое автоматически вызывается при генерации прерывания таймера 08*h*. Таким образом, чтобы не вмешиваться в сложную обработку системой прерывания 08*h*, можно переопределить вектор «заглушки» 1С*h*. Тогда обработчик, «повешенный» на «заглушку», будет выполняться приблизительно 18 раз в секунду.

В принципе, можно было бы не запоминать адрес прерывания, связанного с «заглушкой» и не передавать на это прерывание управление из нового обработчика, поскольку с ней не связано никакое существенное системное прерывание. Однако подавляющее большинство прикладных программ также использует переопределение этого прерывания и, следовательно, они будут несовместимы с написанным таким образом резидентом.

Приведем программу, резидентная часть которой постоянно отображает в левом верхнем углу экрана строку, заданную как параметр программы при загрузке.

```
        .model  tiny
        .code
        .186                            ;Для команд PUSHA–POPA
        org     100h
;НАЧАЛО РЕЗИДЕНТНОЙ ЧАСТИ
InstallEntry:
        jmp     Install                 ;Обход резидентной части
NewHandler      proc
        pusha                           ;Сохраняем все регистры
        push    DS
        push    ES
        push    CS          ;В начале резидента известен только CS
        pop     DS                      ;Настроим регистр DS
        mov     AX,0B800h
        mov     ES,AX       ;ES настроен на начало видеопамяти
        xor     CX,CX
        mov     CL,n
        shl     CX,1                    ;Строка длины 2n символов
        lea     SI,param
        xor     DI,DI
        cld                 ;Прямое направление обработки строк
rep     movsb                   ;Копируем строку из DS:SI в ES:DI
        pop     ES
        pop     DS
        popa
        jmp     CS:OldHandler           ;На старый обработчик
OldHandler      dd      ?               ;Адрес старого прерывания
n       db      ?                       ;Число символов в параметре
param   db      40 dup (?)              ;Выводимая строка
NewHandler      endp
;КОНЕЦ РЕЗИДЕНТНОЙ ЧАСТИ

;НАЧАЛО ИНСТАЛЛЯЦИОННОЙ ЧАСТИ
Install:        mov     AL,ES:80h
        cmp     AL,1                    ;Есть параметр?
        jbe     NoPar                   ;Нет
```

```
        dec     AL    ;«Возврат каретки» не включаем в строку
        mov     n,AL
        xor     SI,SI
        xor     DI,DI
        xor     CX,CX
        mov     CL,n
RPT:    mov     AL,ES:[82h+SI]
        mov     [param+DI],AL
        mov    byte ptr [param+DI+1],0F4h    ;Атр. симв.
        inc     SI
        add     DI,2
        loop     RPT
        mov     AX,351Ch
        int     21h                   ;Старое значение вектора
        mov     word ptr OldHandler,BX
        mov     word ptr OldHandler+2,ES
        mov     AX,251Ch
        mov     DX,offset NewHandler
        int     21h                   ;Новое значение вектора
        mov     DX,offset Install     ;Длина резид. части
        int     27h              ;Завершить и оставить в памяти
NoPar:  mov     AH,09h             ;У программы параметра нет
        lea     DX,Msg
        int     21h
        ret               ;Обычное завершение COM-программы
Msg db 10,13,'USAGE: timer String',10,13,7,'$'
        end     InstallEntry
;КОНЕЦ ИНСТАЛЛЯЦИОННОЙ ЧАСТИ
```

6. Основные понятия защищенного режима

Разработка защищенного режима преследовала следующие основные цели:

- обеспечение многозадачности,
- защита кодов и данных одновременно выполняемых программ друг от друга,
- расширение адресного пространства,
- возможность работы с непрерывными массивами данных длины более 64 К.

С точки зрения программиста, фундаментальное различие реального и защищенного режимов состоит в следующем.

В реальном режиме операционная система только загружает программу в память и передает ей управление, а далее компьютер работает полностью *под управлением программы пользователя* (за исключением прерываний, когда операционная система временно берет управление на себя). Например, нетрудно написать программу без прерываний, в выполнение которой операционная система совершенно не будет вмешиваться.

В защищенном режиме программа пользователя, наоборот, все время работает *совместно* с операционной системой. В частности, ресурсов процессора уже не хватает на обеспечение механизма адресации, и в нем участвует операционная система.

Если бы мы захотели написать программу, которая монопольно управляет работой компьютера в защищенном режиме, нам это скорее всего не удалось бы, поскольку операционная система «не согласится» с передачей своих функций управления в программу. Конечно, гипотетической альтернативой здесь могла бы быть программа, частью которой является некоторая «кустарная» операционная система, поддерживающая защищенный режим.

6.1. Принципы адресации в защищенном режиме

Рассмотрим систему адресации и адресное пространство защищенного режима.

Адрес по-прежнему состоит из сегмента и смещения, однако:

1) смещение может быть как 16-разрядным, так и 32-разрядным,

2) размер сегмента ограничивается максимальной величиной смещения, т. е. величиной *FFFF FFFF + 1* = 2^{32} = *4* Гбайт,

3) адрес сегмента недоступен для программиста и не указывается в программе.

В сегментных регистрах (которые по-прежнему 16-разрядные) содержатся так называемые *селекторы*. Биты 0 и 1 селектора определяют уровень привилегий при доступе к запрашиваемому сегменту, бит 2 определяет, по какой из двух таблиц (*LDT* или *GDT* — см. ниже) будет устанавливаться соответствие между селектором и сегментом. Остальные биты селектора 3–15 рассматриваются как номер дескриптора в одной из двух таблиц дескрипторов *LDT/GDT* (*Local/Global Descriptor Table*).

Таблицы дескрипторов создаются операционной системой и программа не имеет к ним прямого доступа (за исключением указания селекторов).

Каждый элемент таблицы дескрипторов, т. е. каждый *дескриптор*, имеет объем 8 байт и содержит информацию по отдельному сегменту, прежде всего его адрес и длину, которые в этом случае носят название *базы* и *границы*. База — 32-разрядный адрес начала сегмента, граница — 20-разрядный размер сегмента (единица измерения может быть либо байт, либо 4096 байт в зависимости от значения других полей дескриптора).

Кроме того, в дескрипторе содержится много другой информации о сегменте: различные правила умолчания при работе с ним, 32/16-разрядный тип смещений, тип и принадлежность сегмента, уровень привилегий при доступе, режим доступа (чтение/запись/выполнение) и проч.

Для каждой задачи имеются две таблицы дескрипторов — глобальная таблица *GDT* (одна) и локальная таблица *LDT* (по одной на каждую задачу).

Итак, если мы указываем адрес в виде СЕЛЕКТОР:СМЕЩЕНИЕ, по значению селектора в соответствующей таблице дескрипторов отыскивается сегмент, и физический адрес получается как

сумма базы и смещения (в том случае, когда не действует более сложный режим страничной организации памяти).

Подчеркнем, что установление связи между селекторами и сегментами выполняет операционная система и программист не может знать место расположения сегментов в физической памяти.

Подсчитаем максимальный размер адресного пространства при таком механизме адресации. Поскольку на индекс дескриптора в селекторе отводится 13 бит, в двух таблицах (*GDT* и *LDT*) может быть максимально $2\times2^{13} = 2^{14}$ дескрипторов. Размер каждого сегмента ограничен максимальной величиной 32-разрядного смещения, т. е. 2^{32}. Отсюда максимальный размер адресного пространства составит фантастическую величину $2^{14}\times2^{32} = 2^{46} =$ 64 терабайт (1 терабайт = 1024 гигабайт).

Однако современный компьютер *IBM* с 32-разрядной адресной шиной максимально может адресовать только 2^{32} = 4 Гбайт памяти. Адресацию большего объема памяти можно имитировать подкачкой необходимых сегментов с жесткого диска. Кстати, многие операционные системы так и поступают по другой причине — при недостаточном объеме установленной на компьютере физической памяти.

Конечно, сложный механизм адресации защищенного режима обеспечивает большую гибкость в распределении сегментов. Однако в одном случае можно получить адресацию даже более простую, чем в реальном режиме. Допустим, что базы всех сегментов установлены в ноль, разрядность всех смещений равна 32, а границы всех сегментов установлены в 4 Гбайт. Тогда получим так называемую модель памяти **FLAT**, в которой доступ к памяти осуществляется только указанием 32-разрядных смещений. Можно считать, что в этом случае сегментов вообще нет (в этом смысле модель *FLAT* часто называют бессегментной).

Однако модель *FLAT* обладает тем недостатком, что в ней невозможно защитить отдельные участки памяти. В целях такой защиты может применяться режим страничной организации *памяти (pagination)*.

При такой организации вся память разбивается на *страницы*, которые представляют собой непрерывные участки памяти

длиной 1000*h* = *4096* = *4* Кб (*Pentium II* может поддерживать и страницы по 4 Мб).

Адресация при страничном режиме двухуровневая. Данные по 1024 страницам сведены в таблицу с 4-байтными элементами (кстати, такая таблица также является также страницей). Число таких таблиц может доходить до 1024. Данные по всем таблицам также сведены в одну таблицу с 4-байтными элементами, которую можно назвать главным каталогом таблиц страниц. Таким образом, страничная организация охватывает максимально 1024×1024×4096 = 4 Гб.

При страничной организации 32-разрядное смещение рассматривается следующим образом. Биты 31–22 — номер таблицы страниц в каталоге, биты 21–12 — номер страницы в таблице страниц, биты 11-0 — смещение от начала страницы. Для ускорения выбора страниц предусмотрено кеширование, т. е. хранение в буфере адресов недавно использованных страниц.

Страничная организация памяти позволяет решить две следующие основные задачи.

1. *Защита* отдельных страниц (или даже целых групп страниц) путем указания в элементах таблицы страниц (или в элементах каталога таблиц) приблизительно той же информации, что и в дескрипторе сегмента (см. выше).

2. Организация *расширенной виртуальной памяти* путем подкачки с жесткого диска нужных страниц.

Защита как сегментов, так и страниц осуществляется следующим образом. Каждому сегменту или странице присваивается уровень привилегий, который указывается в дескрипторах сегментов и в элементах страничных таблиц. Каждой программе также присваивается некоторый уровень привилегий. При каждом обращении к памяти процессор проверяет соответствие привилегий и при его нарушении генерирует исключение.

Кроме того, при обращении к памяти производятся другие многочисленные проверки, например, запрещена запись в сегмент, помеченный как сегмент кода команд. Аналогичная защита предусмотрена и у страниц.

6.2. Регистры для управления защищенным режимом

Выше было упомянуто о расширении регистров общего назначения до 32 разрядов и о введении двух новых дополнительных сегментных регистров *FS* и *GS*. Расширен также регистр флагов *FLAGS* до 32-разрядного регистра *EFLAGS*, причем его новые биты 21–16 играют большую роль в обеспечении многозадачности и защиты.

На самом деле сегментные регистры *CS*, *DS*, *SS*, *ES*, *FS* и *GS* также расширены до 10 байт, но старшие 8 байт остаются недоступными для программиста. Иногда эти 8-байтные ячейки считаются отдельными регистрами и *называются теневыми регистрами* соответствующих сегментных регистров. При загрузке селектора в сегментный регистр процессор загружает в старшие 8 байт весь дескриптор, соответствующий этому селектору и далее уже не обращается к таблице дескрипторов. Кстати, в старших моделях процессора и при работе в реальном режиме в этих 8 байтах формируется структура, аналогичная дескриптору, хотя никакой таблицы дескрипторов нет.

Для обеспечения сегментной адресации (т. е. для работы с таблицами *GDT* и *LDT*) и для обработки прерываний введены регистры *GDTR–6* байт, *IDTR–6* байт, *LDTR–10* байт, *TR*-10 байт. Восемь старших байт последних двух регистров также предназначены для копирования дескрипторов и программисту недоступны.

Для обеспечения страничной адресации и защиты, а также для управления процессом кеширования введены пять 32-разрядных регистров *CR0–CR4*.

Для целей отладки введены восемь 32-разрядных регистров *DR0–DR7*.

Кроме того, в различные модели процессора *Intel* вводятся различные регистры для оптимизации процессов кеширования, сбора статистической информации о работе процессора и обращениях к памяти. Таких регистров могут быть десятки.

Для управления новыми регистрами введено более 30 новых команд, которые могут выполняться только в защищенном режиме.

6.3. Исключения и прерывания

Трактовка исключений и прерываний в защищенном режиме похожа на их трактовку в реальном режиме с тем отличием, что вместо таблицы векторов прерываний, находящейся на фиксированном месте (в начале) памяти, *таблица дескрипторов прерываний IDT* может быть расположена в произвольном месте в памяти, а ее адрес хранится в специально предназначенном для этого регистре процессора *IDTR*.

Напомним, что исключения могут быть внутренние, программные и внешние. Внутренние прерывания или исключения (*exceptions*) возникают в процессоре при возникновении в нем конфликтной ситуации (например, деления на ноль). Внешние прерывания поступают от внешних устройств. Программные прерывания инициируются самой программой (по команде *int*). В защищенном режиме в режиме многозадачности возникает четвертая разновидность прерываний — переключение с одной задачи на другую.

Несмотря на разную природу прерываний, методы их обработки и в реальном, и в защищенном режимах совпадают — происходит передача управления на соответствующую процедуру обработки прерывания.

Общий сценарий обработки прерывания таков. В микропроцессоре возникает исключение, или исполняемая программа генерирует прерывание, или прерывание поступает от внешней аппаратуры на вход прерываний микропроцессора *INT*. Далее все происходит одинаково. Процессор определяет номер прерывания и дескриптор прерывания, соответствующий этому номеру. Дескриптор прерывания содержит, помимо всего прочего, адрес обработчика прерывания. Далее в стеке сохраняется адрес возврата и управление передается на обработчик, который должен заканчиваться командой **IRET**.

По аналогии с реальным режимом все прерывания защищенного режима занумерованы (от 0 до 255), причем первые 32 номера закреплены именно за исключениями (в настоящее время используется только несколько первых номеров). Вся информация о прерываниях собрана в таблицу *IDT*, элементы которой называются дескрипторами прерываний, или *шлюзами*.

Дескрипторы прерываний, как и дескрипторы сегментов, имеют длину 8 байт, но структура их различна. Точнее можно сказать, что шлюзы являются стандартными, зарезервированными системой дескрипторами сегментов.

Соответственно указанным типам прерываний шлюзы, входящие в таблицу, могут быть трех типов: шлюзы задач — для переключения задач, шлюзы прерываний — для аппаратных прерываний и исключений, и шлюзы ловушек (*trap*) для программных прерываний. Тип данного шлюза указывается двумя байтами внутри кода этого шлюза.

В шлюзах, соответствующих программным прерываниям, указывается уровень привилегий программы, которая может пользоваться данным прерыванием.

Приложение

Учебное расширение языка для операций ввода/вывода

Как мы видели, в отличие от языков высокого уровня операции ввода и вывода чисел, символов и строк на Ассемблере не являются простыми, в особенности для первоначального этапа обучения.

Для облегчения выполнения упражнений на первоначальном этапе обучения можно написать процедуры наиболее употребительных операций ввода/вывода и завершения программы. Можно воспользоваться пакетом процедур учебного расширения, приведенным в [1]. Для экономии места мы не будем здесь приводить текст этих или аналогичных им процедур. Процедуры записаны в файл IOPROC.ASM, его следует заново откомпилировать на имеющейся версии компилятора Ассемблера и получить файл IOPROC.OBJ.

Для большей наглядности и удобства пользования обращения к этим процедурам оформлены в виде макросов, внешне напоминающих обращения к процедурам ввода/вывода на языках высокого уровня. Макроопределения этих макросов собраны в файл IO.ASM, включаемый по директиве INCLUDE.

Таким образом, для использования процедур ввода/вывода необходимо: во-первых, включить файл макроопределений IO.ASM при помощи директивы INCLUDE вместе (или вместо) с собственными макроопределениями, во-вторых, скомпоновать OBJ-файл, полученный вместе с файлом IOPROC.OBJ.

Например, для компоновщика Ассемблера TASM команда компоновки будет выглядеть так:

tlink MYPROG.OBJ+IOPROC.OBJ

Макросы файла IO.ASM, их назначение и описание их операндов приведены в следующей таблице.

Обращение	Назначение	Операнды	Примечания
ОПЕРАЦИИ ВВОДА			
flush	Очистка буфера клавиатуры	Нет	
inch x	Ввод символа с клавиатуры	РОН-8, байт	После ввода нажать Enter
inint x	Ввод целого со знаком и без знака	РОН-16, слово	После ввода нажать Enter Диапазон: -32768..65535
ОПЕРАЦИИ ВЫВОДА			
newline	Переход на новую строку	Нет	
outch x	Вывод символа на экран	РОН-8, байт, символ, ASCII-код	
outstr	Вывод строки на экран	Нет	Строка по адресу DS:DX. В конце строки должен быть дополнительный символ $
outint x,l	Ввод целого со знаком	*x*: РОН-16, слово, число *l*: РОН-8, байт, число	*x* - выводимое число *l* - число позиций (необязательный операнд)
outword x,l	Ввод целого без знака	*x*: РОН-16, слово, число *l*: РОН-8, байт, число	*x* - выводимое число *l* - число позиций (необязательный операнд)
ЗАВЕРШЕНИЕ ПРОГРАММЫ			
finish	Завершение программы	Нет	В регистр AL можно записать код завершения, который будет передан в DOS

СПИСОК ЛИТЕРАТУРЫ

1. *Пильщиков В.Н.*. Программирование на языке ассемблера IBM PC. — М.: Диалог-МИФИ, 1999.
2. *Шереметьев К.* Введение в Турбо-Ассемблер. — М.: Либрис, 1993.
3. *Зубков С.В.*. Assembler для DOS, Windows и UNIX. — М.: ДМК, 1999.
4. *Майко Г.В.*. Assembler для IBM PC. — М.: Бизнесинформ. 1999.
5. *Финогенов К.Г.*. Основы языка Ассемблера. — М.: Радио и связь, 1999.
6. *Рудаков П.И., Финогенов К.Г.*. Программируем на языке ассемблера IBM PC. — Обнинск. Принтер. 1999.
7. *Джордейн Р.* Справочник программиста персонального компьютера IBM PC. — М.: Финансы и статистика, 1992.
8. *Юров В., Хорошенко С.* Assembler. Учебный курс. — СПб: ПИТЕР, 1999.
9. *Абель П.* Язык ассемблера для IBM PC и программирования. — М.: Высшая школа, 1992.
10. *Нортон П., Соухэ Д.* Язык ассемблера для IBM PC. — М.: Финансы и статистика. 1992.
11. *Сван Т.* Освоение Turbo Assembler. — М.: Диалектика, 1996.
12. *Графические адаптеры EGA и VGA*. Руководство по программированию. — М.: Программпродукт, 1992.

Оглавление

Предисловие. Зачем нужен Ассемблер? 3

1. Архитектурные особенности процессоров Intel™ 5

1.1. Представление целых чисел в процессоре и в памяти 5

1.1.1. Применяемые системы счисления 5

1.1.2. Биты, байты, слова, двойные слова.............8

1.1.3. Беззнаковые и знаковые величины. Прямой и дополнительный коды.............. .10

1.1.4. Двоично-десятичная кодировка (BDC) 11

1.2. Реальный режим (80x86) 12

1.2.1. Понятие о реальном и защищенном режимах 12

1.2.2. Регистры процессора и их назначение.........12

1.2.3. Развитие архитектуры процессора Intel 14

1.2.4. Адресное пространство процессора. Физические адреса, сегменты, смещения.....15

1.2.5. Типичная схема адресного пространства 80x86 18

2. Язык программирования Ассемблера 20

2.1. Процесс программирования и выполнения программы...20

2.2. Синтаксис ассемблерной программы 22

2.3. Простейшая программа 23

2.4. "Препроцессорные" директивы INCLUDE и EQU...28

2.5. Директивы описания и инициализации переменных DB, DW и DD......................................30

2.5.1. Переменная и ее атрибуты 30

2.5.2. Выражения 31

2.5.3. Директива DB 32
2.5.4. Директивы DW и DD 33
2.6. Сегментная структура программы и модели памяти 34
2.6.1. Программные сегменты 34
2.6.2. Директивы модели памяти и упрощенного задания программных сегментов 36
2.6.3. Директива изменения счетчика размещения ORG 40
2.6.4. Структура исполняемых файлов. Исполняемые файлы EXE и COM 40
2.7. Команды Ассемблера 44
2.7.1. Общие сведения о командах 44
2.7.2. Адресация 46
2.7.2.1. Прямая адресация 47
2.7.2.2. Косвенная адресация 48
2.7.3. Команды пересылки и преобразования 49
2.7.3.1. Команда пересылки MOV 50
2.7.3.2. Команды условной пересылки CMOVxx 51
2.7.3.3. Команда обмена операндов XCHG 52
2.7.3.4. Команда обмена байт в 32-регистре BSWAP 52
2.7.3.5. Команда конвертирования байта в слово CBW 52
2.7.3.6. Команды конвертирования слова в двойное слово CWD и CWDE 53
2.7.3.7. Команда перекодировки в соответствии с таблицей XLAT 53
2.7.3.8. Команда загрузки исполнительного адреса LEA 54
2.7.4. Команды двоичной арифметики 54
2.7.4.1. Команды сложения и вычитания ADD, SUB, ADC, SBB, INC, DEC, CMP 54
2.7.4.2. Команды умножения и деления MUL,DIV,IMUL,IDIV 56

2.7.5. Команды передачи управления и циклы 57
2.7.5.1. Безусловный переход JMP 58
2.7.5.2. Условные переходы Jxxx 59
2.7.5.3. Команды циклы LOOP, LOOPZ, LOOPNZ60
2.7.5.4. Команды вызова процедуры CALL и возврата из процедуры RET *61*
2.7.6. Команды работы со стеком 62
2.7.6.1. Команды вталкивания в стек PUSH и выталкивания из стека POP 64
2.7.6.2. Команды сохранения в стеке регистра флагов PUSHF и POPF 65
2.7.6.3. Команды сохранения в стеке регистров общего назначения PUSHA и POPA 65
2.7.6.4. Работа со стеком, как с обычной памятью . 65
2.7.7. Логические команды 66
2.7.7.1. Команды логического умножения (конъюнкции) AND и TEST 66
2.7.7.2. Команда логического сложения (дизъюнкции) OR 67
2.7.7.3. Команда исключающего логического "ИЛИ" XOR 67
2.7.7.4. Команда логического отрицания NOT 67
2.7.8. Команды сдвига 67
2.7.8.1. Команды логического сдвига SHL и SHR 68
2.7.8.2. Команды арифметического сдвига SAL и SAR 68
2.7.8.3. Команды циклического сдвига ROL, ROR и RCL, RCR 69
2.7.9. Команды модификации флагов и команда холостого хода 69
2.7.10. Команды обработки цепочек 70
2.7.10.1. Пересылка цепочек *MOVSB, MOVSW* 71
2.7.10.2. Сравнение цепочек *CMPSB, CMPSW* 72
2.7.10.3. Сканирование цепочки *SCASB, SCASW*.. 72

2.7.10.4. Запись значения регистра в элемент цепочки *STOSB* и *STOSW*........................72
2.7.10.5. Загрузка элемента цепочки *LODSB* и *LODSW*73
2.7.11. Процедуры 73
2.7.11.1. Упрощенное описание процедур при помощи конвенций C, PASCAL. CTDCALL 74
2.7.12. Команды ввода/вывода. Прерывания 78
2.7.12.1. Команды ввода и вывода IN и OUT....... 78
2.7.12.2. Концепция прерывания 78
2.7.12.3. Функции DOS-прерывания 21*h* и BIOS-прерывания 10*h* 82
2.7.13. Макросы и макроопределения 85
Приемы программирования на Ассемблере......................91
3.1. Массивы и структуры 91
3.1.1. Работа с массивами........................92
3.1.2. Структуры 96
3.2. Реализация конструкций языков высокого уровня 98
3.2.1. Логические выражения........................98
3.2.2. Конструкция IF-THEN-ELSE и переключатель........................99
3.2.3. Циклы 101
3.2.4. Программы с параметрами *103*
Работа с внешними устройствами 104
4.1. Видеопамять и работа с экраном 104
4.1.1. Текстовый режим 104
4.1.2. Графический режим 109
4.2. Клавиатура 111
4.3. Мышь........................113
4.4. Файлы 114
4.4.1. Создание, открытие и закрытие файла...... 115
4.4.2. Чтение из файла и запись в него 117

5. Резиденты. Перехватчики. Обработка событий 120
5.1. Резидентные программы 120
5.2. Обработчики прерываний 122
6. Основные понятия защищенного режима 133
6.1. Принципы адресации в защищенном режиме *133*
6.2. Регистры для управления защищенным режимом *137*
6.3. Исключения и прерывания *138*
Приложение. Учебное расширение языка для операций ввода/вывода 140
Список литературы 142
Оглавление 143

Учебное издание

Митницкий Василий Яковлевич

АРХИТЕКТУРА IBM PC И ЯЗЫК АССЕМБЛЕРА

Редактор *И.А. Волкова*
Корректор *О.П. Котова*

Изд. лиц. № 040060 от 21.08.96. Подписано в печать 10.04.2000.
Формат 60 x 84 1/16. Бумага офсетная. Печать офсетная.
Усл. печ. л. 9,25. Уч.-изд. л. 8,7. Тираж 2000 экз. Заказ № 182.

Московский физико-технический институт
(государственный университет)

Отдел автоматизированных издательских систем
"ФИЗТЕХ-ПОЛИГРАФ"
141700, Московская обл., г. Долгопрудный, Институтский пер., 9